Introdução à Computação

por Gilberto Farias e Eduardo Santana Medeiros

Ed. v1.0

COLLABORATORS

	TITLE : Introdução à Computação		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Gilberto Farias e Eduardo Santana Medeiros	20 de maio de 2013	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
v1.0	Março 2013	Primeira versão do livro	Gilberto Farias, Eduardo Santana

Sumário

Prefácio

texto

Processo ágil de Produção de Livros

O processo é **ágil** e este curso *também* será.

Importante
Leia com atenção os objetivos e a descrição do curso antes de se inscrever. Quer saber mais? Consulte "Processo ágil de Produção de Livros" [vii].

- Fazer lista é fácil
- Incluir referências é brincadeira de criança

Exemplo de como inserir códigos no texto

```
#include "teste.h"
#include <stdio.h>

int main(){
  int a, b;
  a = soma(2, 3);
  printf("Soma = %d\n", a);
  b = subtrai(4, 3);
  printf("Subtração = %d\n", b);
}
```

Público alvo

O público alvo desse livro são os alunos de Licenciatura em Computação, na modalidade à distância [1]. Ele foi concebido para ser utilizado numa disciplina de *Introdução à Computação*, no primeiro semestre do curso.

[1] Embora ele tenha sido feito para atender aos alunos da Universidade Federal da Paraíba, o seu uso não se restringe a esta universidade, podendo ser adotado por outras universidades que adotam o sistema UAB.

Método de Elaboração

Este livro foi realizado com Financiamento da CAPES.

Como você deve estudar cada capítulo

- Leia a visão geral do capítulo
- Estude os conteúdos das seções
- Realize as atividades no final do capítulo
- Verifique se você atingiu os objetivos do capítulo

NA SALA DE AULA DO CURSO

- Tire dúvidas e discuta sobre as atividades do livro com outros integrantes do curso
- Leia materiais complementares eventualmente disponibilizados
- Realize as atividades propostas pelo professor da disciplina

Caixas de diálogo

Durante o texto foram colocadas caixas de diálogo, nesta seção apresentamos os significados delas.

Nota
Esta caixa é utilizada para realizar alguma reflexão.

Dica
Esta caixa é utilizada quando desejamos remeter a materiais complementares.

Importante
Esta caixa é utilizada para chamar atenção sobre algo importante.

Cuidado
Esta caixa é utilizada para alertar sobre algo potencialmente perigoso.

Atenção
Esta caixa é utilizada para alertar sobre algo potencialmente perigoso.

Os significados das caixas são apenas uma referência, podendo ser adaptados conforme as intenções dos autores.

Vídeos

Os vídeos são apresentados da seguinte forma:

Figura 1: Exemplo de vídeo: http://youtu.be/uDMs-TyjSek

Nota
Na **versão impressa** irá aparecer uma imagem quadriculada. Isto é o qrcode (http://pt.wikipedia.org/wiki/C%C3%B3digo_QR) contendo o link do vídeo. Caso você tenha um celular com acesso a internet poderá acionar um programa de leitura de qrcode para acessar o vídeo.
Na **versão digital** você poderá assistir o vídeo clicando diretamente sobre o link ou acionando o play (na versão em HTML).

Compreendendo as referências

Durante o texto nós podemos ter várias referências:

Referências a capítulos
Prefácio [vii]

Referências a seções
“Como você deve estudar cada capítulo” [viii], “Caixas de diálogo” [viii].

Referências a imagens e tabelas
Figura 2 [xii] Tabela 1 [xi]

Nota
Na **versão impressa**, o número que aparece entre chaves “[ ]” corresponde ao número da página onde se entra o conteúdo referenciado. Nas **versões digitais** do livro você poderá clicar no link da referência.

Códigos e comandos

Os códigos ou comandos são apresentados com a seguinte formação:

```
cc -S main.c teste.c
```

No exemplo a seguir, temos outra apresentação de código fonte. Desta vez de um arquivo `main.c`, que se encontra dentro do diretório `code/tradutor`. O diretório `tradutor` faz referência ao capítulo onde o código será apresentado.

code/tradutor/main.c

```
#include "teste.h"
#include <stdio.h>

int main(){
  int a, b;
  a = soma(2, 3);
  printf("Soma = %d\n", a);
  b = subtrai(4, 3);
  printf("Subtração = %d\n", b);
}
```

Baixando os códigos fontes

Existem duas formas de acessar os códigos fontes contidos neste livro.

Acesso on-line individual
Você pode acessar individualmente os arquivos deste livro pelo endereço: https://github.com/-edusantana/introducao-a-computacao-livro/tree/master/livro/capitulos/code.

Baixando todos os códigos
Você também pode baixar o código fonte do livro inteiro, que contém todos os códigos mencionados no livro. Existem duas formas de baixar o código inteiro, através de um arquivo zip ou clonando o repositório.

Arquivo zip
https://github.com/edusantana/introducao-a-computacao-livro/archive/master.zip. Depois de baixar o arquivo, descompacte-o.

Clonando o repositório
Use o comando: git clone https://github.com/edusantana/introducao-a-computacao-livro

Nota
Independente do método utilizado para acessar os arquivos, os códigos fontes estão organizados por capítulos no diretório `livro/capitulos/code`.

Atenção
Os códigos acessados por estes métodos são referentes à versão mais nova do livro (em produção). É possível que eles sejam diferentes da versão que você tenha no impresso.

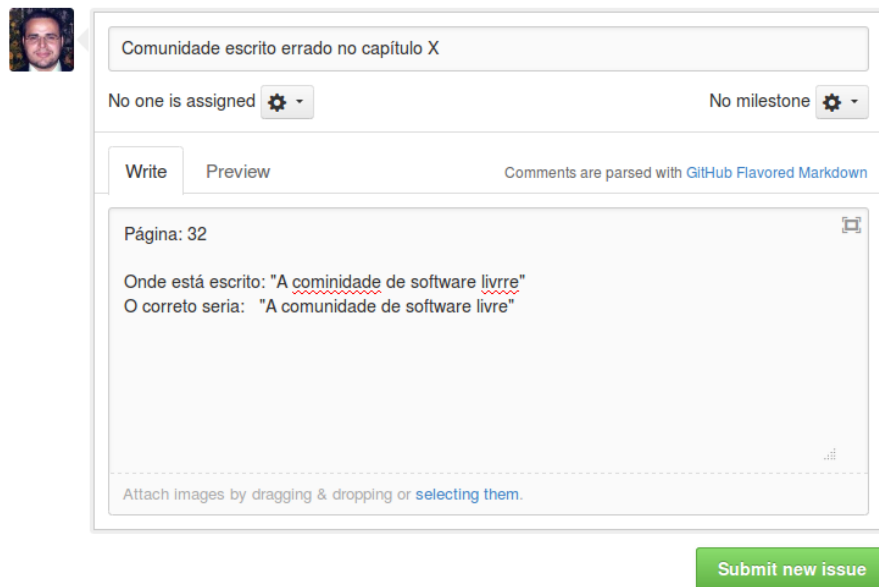
Contribuindo com o livro

Você pode contribuir com a atualização e correção deste livro. A tabela a seguir resume os métodos de contribuições disponíveis:

Tabela 1: Métodos para contribuição do livro

Método de contribuição	Habilidades necessárias	Descrição
Issue track	• Inscrição no site do github • Preenchimento de um formulário	Consiste em acessar o repositório do livro e submeter um erro, uma sugestão ou uma crítica — através da criação de um *Issue*. Quando providências forem tomadas você será notificado disso.
Submissão de correção	• Realizar fork de projetos • Atualizar texto do livro • Realizar PullRequest	Consiste em acessar os arquivos fontes do livro, realizar a correção desejada e submetê-la para avaliação. Este processo é o mesmo utilizado na produção de softwares livres.

Contribuição através do Issue track
Para contribuir com um erro, sugestão ou crítica através de um envio de uma mensagem acesse: https://github.com/edusantana/introducao-a-computacao-livro/issues/new

Figura 2: Exemplo de contribuição através do *Issue track*

Manual sobre produção dos livros

Caso você deseja submeter correções, precisará compreender o processo de produção dos livros. O manual que o descreve pode ser acessado em http://producao.virtual.ufpb.br.

Capítulo 1

História dos computadores

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Citar os precursores dos computadores
- Discorrer sobre a importância do surgimento dos cartões perfurados
- Descrever como eram os computadores em cada uma das 5 gerações
- Explicar porque a palavra *bug* passou a ser empregada para designar defeitos
- Relatar a importância do circuito integrado no processo de fabricação dos computadores

Os computadores fazem parte do dia a dia da sociedade contemporânea, mas você conhece a história deles?

Conhecer a história dos computadores é importante pois é através do estudo do passado que podemos compreender e valorizar o presente. Ao decorrer do capítulo veremos exemplos de como ideias simples contribuíram para evolução da humanidade.

Mas o que é um computador? A palavra computador significa *aquele que faz cálculos*, seja ele pessoa ou máquina. Sem dúvida as pessoas foram os primeiros computadores, já que passavam horas realizando contas e mais contas. Inclusive, veremos mais a adiante que o surgimento de uma simples calculadora causou revolta — pois as pessoas tiveram medo de perder seus empregos. Mas não vamos precipitar nossos estudos, vamos começar pelo início.

Nota

Daqui e em diante, sempre que mencinarmos a palavra *computador* estaremos nos referindo ao seu sentido usual, de máquinas.

Neste capítulo iremos conhecer os instrumentos e máquinas precursores dos computadores, e saberemos em qual momento da história surgiram as máquinas programáveis. Em seguida, estudaremos as gerações de computadores, procurando entender a sua evolução.

Para ajudá-lo na leitura que se segue, convidamos a assistir estes vídeos sobre a História do Computador.

Figura 1.1: História do Computador 1: http://youtu.be/Ixgh3AhiL3E

Figura 1.2: História do Computador 2: http://youtu.be/dWiUZsoLD0M

1.1 Precursores dos computadores

São considerados precursores dos computadores todos os instrumentos ou máquinas que contribuíram com ideias para a criação dos mesmos. Dentre eles, o surgimento de uma máquina programável foi um grande marco na história dos computadores.

Vamos iniciar nossos estudos com um instrumento que talvez você conheça e provavelmente já utilizou na escola, o ábaco.

1.1.1 Ábaco

O ábaco foi um dos primeiros instrumentos desenvolvidos para auxiliar os humanos na realização de cálculos. Muitos atribuem sua criação à China, mas existem evidências deles na Babilônia no ano 300 A.C.

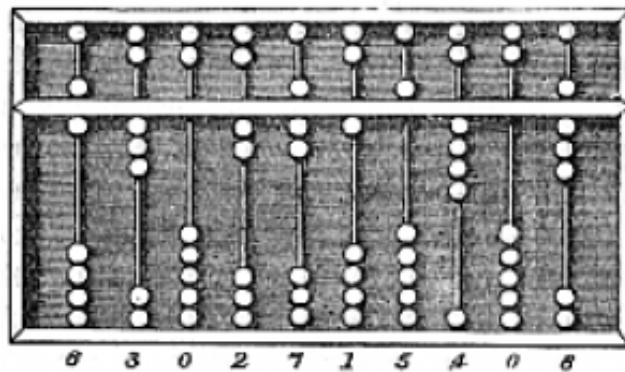



Figura 1.3: Ilustração de um ábaco

A ideia básico no ábaco é considerar as contas (bolinhas) contidas na parte inferior, chamada de chão do Ábaco, com valor unitário e cada conta contida na parte superior, chamada de céu do Ábaco, com valor de cinco unidades. Cada valor unitário tem representação diferente dependendo da coluna em que se encontra, logo, uma unidade na primeira coluna tem valor 1 em nosso sistema numérico, já uma unidade na segunda coluna tem valor 10.

Dica
Você pode conhecer mais sobre o ábaco no seguinte site: http://www.educacaopublica.rj.gov.br/oficinas/matematica/abaco/03.html

1.1.2 Ossos de Napier

Em 1614, John Napier (lê-se Neper) descobriu os cálculos logaritmos.

> "A invenção dos logaritmos surgiu no mundo como um relâmpago. Nenhum trabalho prévio anunciava ou fazia prever a sua chegada. Surge isolada e abruptamente no pensamento humano sem que se possa considerar consequência de obras ou de pesquisas anteriores"
>
> — Lord Moulton

Napier também inventou o que ficou conhecido por "Ossos de Napier"(Figura 1.4 [4]), que auxiliavam na realização de multiplicações, baseando-se na teoria de logaritmos.

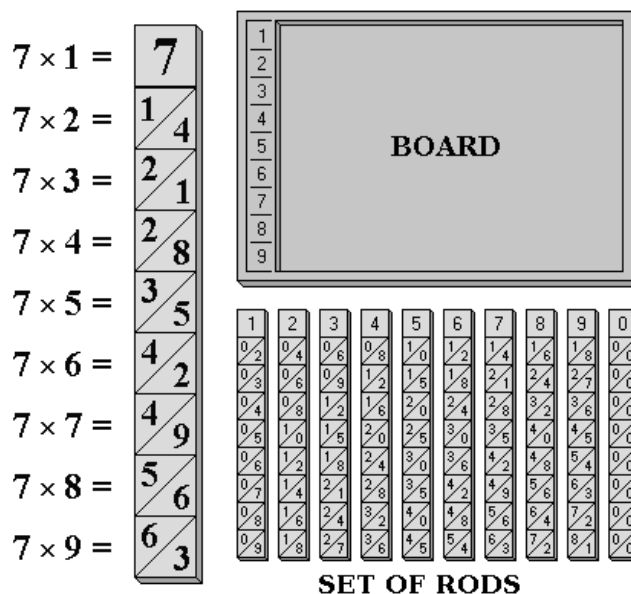



Figura 1.4: Ilustração dos Ossos de Napier.

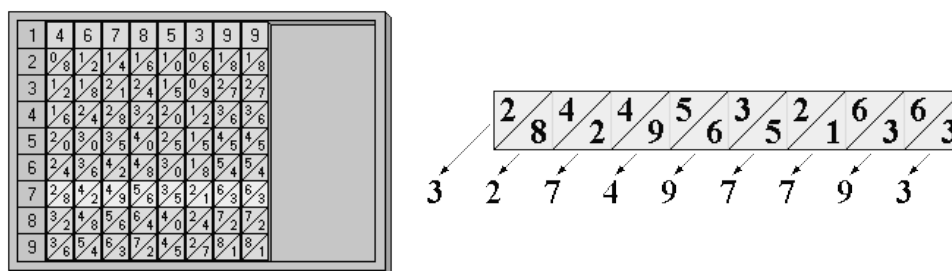



Figura 1.5: Ilustração da operação de multiplicação utilizando os ossos de Napier: 46785399 x 7.

Nota

Para conhecer como os Ossos de Napier funcionam consulte: http://eu-thais.blogspot.com.br/2010/08/como-funcionam-os-bastoes-de-napier.html ou http://en.wikipedia.org/wiki/Napier%27s_bones (em inglês).

A criação da Régua de Cálculo, (Figura 1.6 [4]) em 1632 na Inglaterra, foi diretamente influenciada pelos Ossos de Napier. Esta régua chegou a ser utilizada pelos engenheiros da NASA, na década de 1960, nos programas que levaram o homem à Lua.

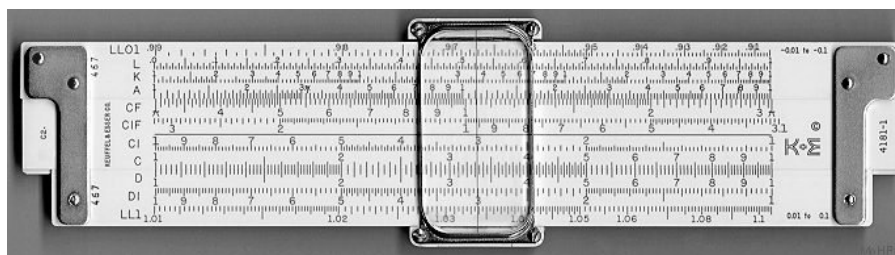



Figura 1.6: Régua de Cálculo

1.1.3 As rodas dentadas de Pascal (Pascaline)

Em 1642, o francês **Blaise Pascal**, aos 19 anos de idade, foi o primeiro a inventar um dispositivo mecânico para realização de cálculos. O dispositivo é conhecido como *As rodas dentadas de Pascal* (ou Pascaline, Figura 1.7 [5]).

Pascal era filho de um cobrador de impostos e auxiliava o pai na realização de cálculos utilizando um instrumento similar ao ábaco. Mas segundo ele, o trabalho era muito entediante, o que o levou a elaborar um dispositivo para realização de somas e subtração.

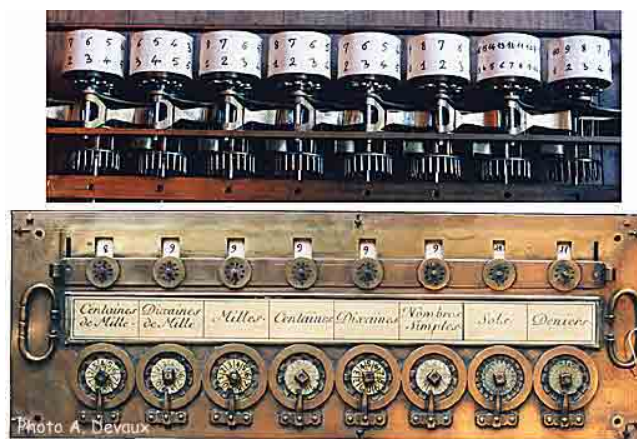
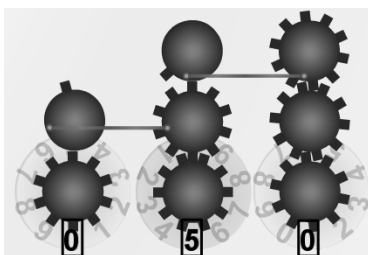



Figura 1.7: Pascaline de 8 dígitos aberta, mostrando as engrenagens (acima), e a apresentação da máquina fechada (abaixo).

O mecanismo de funcionamento é o mesmo utilizado até hoje nos odômetros de carros, onde as engrenagens são organizadas de tal forma a simular o "vai um"para a próxima casa decimal nas operações de adição.

Dica
Existe um animação demonstrando o funcionamento da máquina pascaline, você pode acessá-lo através do seguinte link: http://therese.eveilleau.pagesperso-orange.fr/pages/-truc_mat/textes/pascaline.htm.

As operações de soma eram realizadas girando as engrenagens em um sentido e as operações de subtração no sentido oposto, enquanto que as operações de multiplicação utilizavam vários giros da soma manualmente.

O surgimento da pascaline, no entanto, não agradou a todos, alguns empregados queriam destruir a máquina com medo de perder seus empregos.

Dica
Você pode consultar a biografia de Pascal em: http://www.thocp.net/biographies/-pascal_blaise.html

1.1.4 Leibniz - A primeira calculadora com quatro operações

Em 1672, o Alemão *Gottfried Wilhelm Leibniz* foi o primeiro a inventar uma calculadora que realizava as 4 operações básicas (adição, subtração, multiplicação e divisão). A adição utilizava um mecanismo baseado na Pascaline, mas as operações de multiplicação realizavam a sequência de somas automáticas.

Leibniz também foi o primeiro a defender a utilização do **sistema binário**, que é fundamental nos computadores digitais que utilizamos hoje.

1.1.5 Máquinas Programáveis

Um marco na história foi a invenção de máquinas programáveis, que funcionavam de forma diferente de acordo com uma programação que lhes era fornecida.

1.1.5.1 Tear de Jacquard

Em 1804, o Francês Joseph Marie **Jacquard** inventou uma máquina de tear que trançava o tecido de acordo com uma programação que era fornecida através de furos num cartão.

Figura 1.8: Máquina de tear usando programação através de cartões perfurados.

A invenção de Jacquard revolucionou a industria de tecido, e em 1806, ela foi declarada propriedade pública e ele foi recompensado com uma pensão e *royalties* por cada máquina que fosse construída.

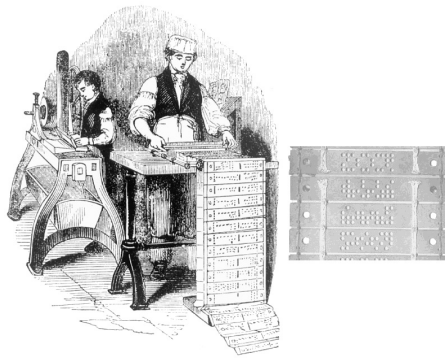



Figura 1.9: Esquerda: Jacquard perfurando os cartões. Direita: cartões perfurados.

1.1.5.2 A Máquina Diferencial

Em 1822, o matemático inglês Charles Babbage propôs a construção de uma máquina de calcular que ocuparia uma sala inteira. O propósito da máquina seria de corrigir os erros das tabelas de logaritmos, muito utilizadas pelo governo britânico devido as grandes navegações. A construção da máquina, no entanto, excedeu em orçamento e tempo na sua construção, foi inclusive o projeto mais caro que o governo britânico já havia financiado. Eventualmente, os subsídios foram retirados e o projeto abortado.

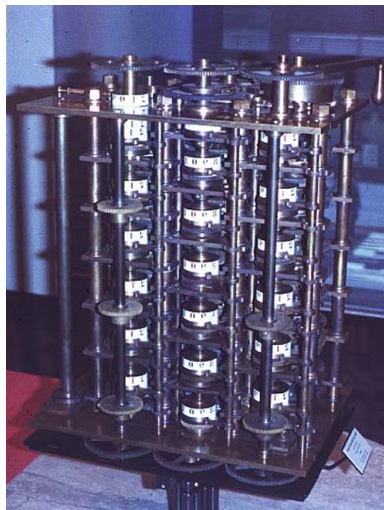



Figura 1.10: Pequena seção da máquina diferencial de *Charles Babbage*.

Vídeo de uma réplica da Máquina Diferencial de Charles Barbage:

Figura 1.11: Demonstração da Máquina Diferencial de Charles Babbage: http://youtu.be/-BlbQsKpq3Ak

1.1.5.3 A Máquina Analítica

Após a inacabada máquina diferencial, em 1837, Charles Babbage anunciou um projeto para construção da *Máquina Analítica*. Influenciado pelo tear de Jacquard, Babbage propôs uma máquina de propósito genérico, utilizando uma programação através de cartões perfurados.

Babbage trouxe um grande avanço intelectual na utilização de cartões perfurados, enquanto Jacquard utilizava os cartões apenas para acionar ou desativar o funcionamento uma determinada seção da máquina de tear, Babbage percebeu que os cartões poderiam ser utilizados para armazenar ideias abstratas, sejam elas instruções ou números, e que poderiam ser referenciados posteriormente, adontando para sua máquina o conceito de memória.

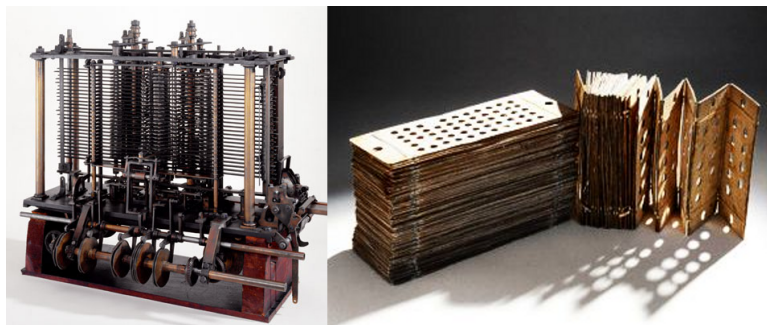



Figura 1.12: Máquina Analítica e os cartões perfurados.

Ele percebeu que os cartões perfurados poderiam ser utilizados para guardar números, sendo utilizados como um mecanismo de armazenamento de dados e futuramente poderiam ser referenciados. Ele idealizou o que hoje chamamos de **unidade de armazenamento** e **unidade de processamento de dados**.

A principal funcionalidade que a diferenciava das máquinas de calcular era a utilização de instruções condicionais. A máquina poderia executar fluxos diferentes baseada em condições que eram avaliadas conforme instruções perfuradas nos cartões.

Nenhum dos dois projetos de Babbage foram concluídos, a máquina analítica se fosse construída teria o tamanho de uma locomotiva.

1.1.5.4 A Primeira programadora

A condessa de Lovelace, Ada Byron, se interessou pela máquina analítica de Babbage e se comunicava com ele através de cartas e encontros. Ela passou a escrever programas que a máquina poderia ser capaz de executar, caso fosse construída. Ela foi a primeira a reconhecer a necessidade de *loops* e sub-rotinas. Por esta contribuição, Ada ficou reconhecida na história como a primeira programadora.

Figura 1.13: Ada Lovelace, primeira programadora.

1.1.6 Linha do tempo

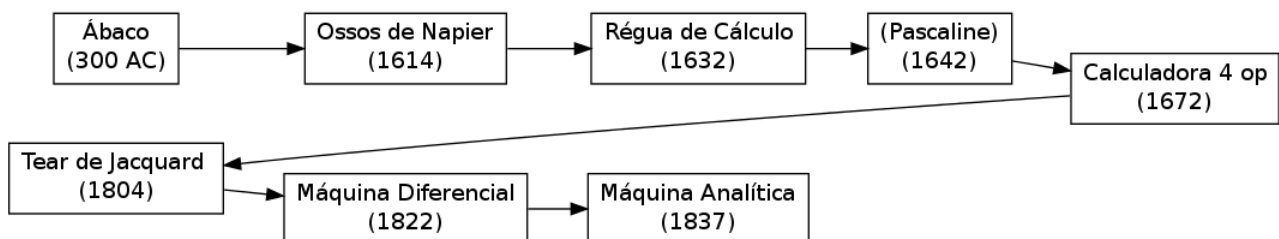



Figura 1.14: Linha do tempo dos precursores dos computadores

1.2 As gerações dos computadores

Os computadores são máquinas capazes de realizar vários cálculos automaticamente, além de possuir dispositivos de armazenamento e de entrada e saída.

Nesta seção iremos ver a evolução dos computadores até os dias atuais.

1.2.1 Primeira Geração (1946-1954)

A primeira geração dos computadores é marcada pela utilização de **válvulas**. A válvula é um tubo de vidro, similar a uma lâmpada fechada sem ar em seu interior, ou seja, um ambiente fechado a vácuo, e contendo eletrodos, cuja finalidade é controlar o fluxo de elétrons. As válvulas aqueciam bastante e costumavam queimar com facilidade.

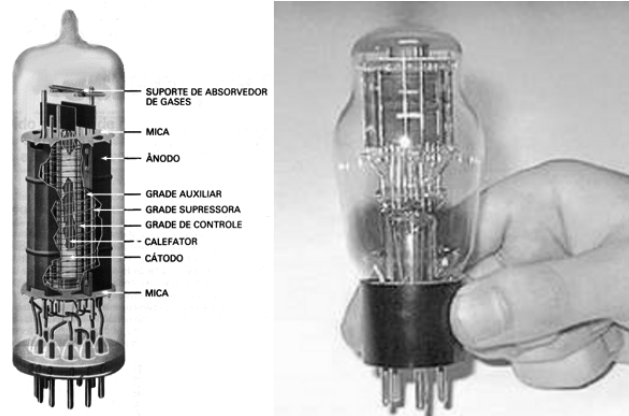



Figura 1.15: As válvulas eram do tamanho de uma lâmpada.

Além disso, a programação era realizada diretamente na linguagem de máquina, o que dificultava a programação e consequentemente despendia muito tempo. O armazenamento dos dados era realizado em cartões perfurados, que depois passaram a ser feitos em fita magnética.

Um dos representantes desta geração é o ENIAC. Ele possuía 17.468 válvulas, pesava 30 toneladas, tinha 180 m^2 de área construída, sua velocidade era da ordem de 100 kHz e possuia apenas 200 bits de memória RAM.

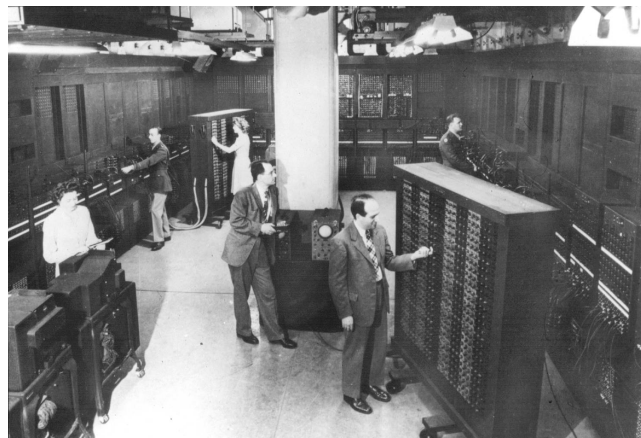



Figura 1.16: ENIAC, representante da primeira geração dos computadores.

Nenhum dos computadores da primeira geração possuíam aplicação comercial, eram utilizados para fins balísticos, predição climática, cálculos de energia atômica e outros fins científicos.

1.2.1.1 Alan Turing - O pai da Ciência da Computação

Alan Mathison Turing(23 de Junho de 1912 — 7 de Junho de 1954) foi um matemático, lógico, criptoanalista e cientista da computação britânico. Foi influente no desenvolvimento da ciência da computação e proporcionou uma formalização do conceito de algoritmo e computação com a máquina de Turing, desempenhando um papel importante na criação do computador moderno. Durante a Segunda Guerra Mundial, *Turing* trabalhou para a inteligência britânica em Bletchley Park, num centro especializado em quebra de códigos. Por um tempo ele foi chefe de Hut 8, a seção responsável

pela criptoanálise da frota naval alemã. Planejou uma série de técnicas para quebrar os códigos alemães, incluindo o método da bombe, uma máquina eletromecânica que poderia encontrar definições para a máquina de criptografia alemã, a Enigma. Após a guerra, trabalhou no Laboratório Nacional de Física do Reino Unido, onde criou um dos primeiros projetos para um computador de programa armazenado, o ACE.

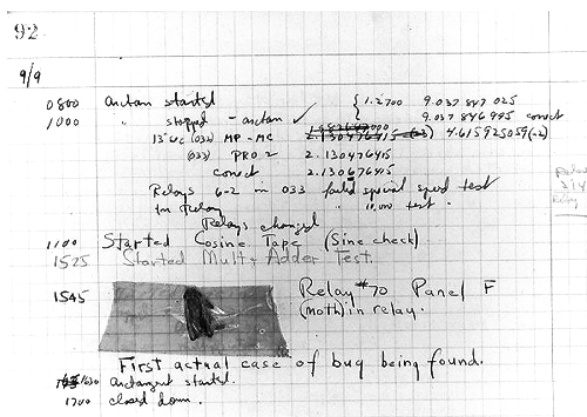
Aos 24 anos de idade, consagrou-se com a projeção de uma máquina que, de acordo com um sistema formal, pudesse fazer operações computacionais. Mostrou como um simples sistema automático poderia manipular símbolos de um sistema de regras próprias. A máquina teórica de *Turing* pode indicar que sistemas poderosos poderiam ser construídos. Tornou possível o processamento de símbolos, ligando a abstração de sistemas cognitivos e a realidade concreta dos números. Isto é buscado até hoje por pesquisadores de sistemas com Inteligência Artificial (IA). Para comprovar a inteligência artificial ou não de um computador, *Turing* desenvolveu um teste que consistia em um operador não poder diferenciar se as respostas a perguntas elaboradas pelo operador eram vindas ou não de um computador. Caso afirmativo, o computador poderia ser considerado como dotado de inteligência artificial. Sua máquina pode ser programada de tal modo que pode imitar qualquer sistema formal. A ideia de computabilidade começou a ser delineada.

A maior parte de seu trabalho foi desenvolvida na área de espionagem e, por isso, somente em 1975 veio a ser considerado o "pai da Ciência da Computação".

Se possível, assista ao vídeo do Globo Ciência sobre a vida e obra de Alan Turing:

Figura 1.17: Vida e Obra de Alan Turing: http://youtu.be/yIluxaHL0v0

O primeiro bug da história

A palavra **bug** (inseto em inglês) é empregada atualmente para designar um defeito, geralmente de software. Mas sua utilização com este sentido remonta a esta época. Conta a história que um dia o computador apresentou defeito. Ao serem investigadas as causas, verificou-se que um inseto havia prejudicado seu funcionamento. A foto abaixo, supostamente, indica a presença do primeiro bug.

Até hoje os insetos costumam invadir os equipamentos eletrônicos, portanto observe-os atentamente, evite deixar comida próximo ao computador e não fique sem utilizá-lo por um longo período.

1.2.2 Segunda Geração (1955-1964)

A segunda geração de computadores foi marcada pela substituição da válvula pelo **transistor**. O transistor revolucionou a eletrônica em geral e os computadores em especial. Eles eram muito menores do que as válvulas a vácuo e tinham outras vantagens: não exigiam tempo de pré-aquecimento, consumiam menos energia, geravam menos calor e eram mais rápidos e confiáveis. No final da década de 50, os transistores foram incorporados aos computadores.

Dica

Para saber mais sobre o funcionamento dos transistores consulte http://pt.wikipedia.org/wiki/-Transistor.

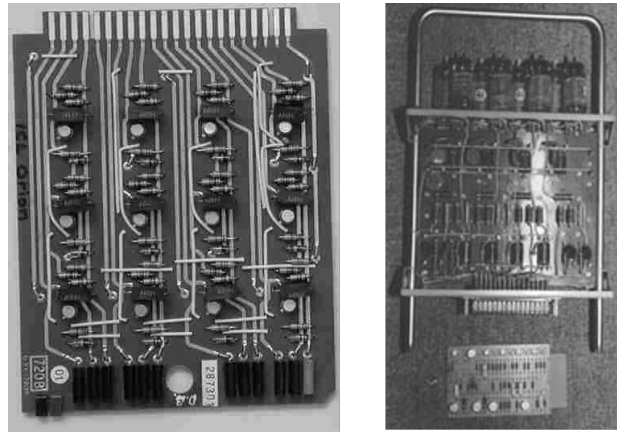



Figura 1.18: Circuito com vários transistores (esquerda). Comparação do circuito com válvulas (canto superior-direito) com um circuito composto de transistores (inferior-direito).

Na segunda geração o conceito de Unidade Central de Procedimento (CPU), memória, linguagem de programação e entrada e saída foram desenvolvidos. O tamanho dos computadores diminuiu consideravelmente. Outro desenvolvimento importante foi a mudança da linguagem de máquina para a linguagem assembly, também conhecida como linguagem simbólica. A linguagem assembly possibilita a utilização de *mnemônicos* para representar as instruções de máquina.

Figura 1.19: Computadores IBM da segunda geração.

Em seguida vieram as linguagens de alto nível, como, por exemplo, Fortran e Cobol. No mesmo período surgiu o armazenamento em disco, complementando os sistemas de fita magnética e possibilitando ao usuário acesso rápido aos dados desejados.

1.2.3 Terceira Geração (1964-1977)

A terceira geração de computadores é marcada pela utilização dos **circuitos integrados**, feitos de silício. Também conhecidos como **microchips**, eles eram construídos integrando um grande número de transistores, o que possibilitou a construção de equipamentos menores e mais baratos.

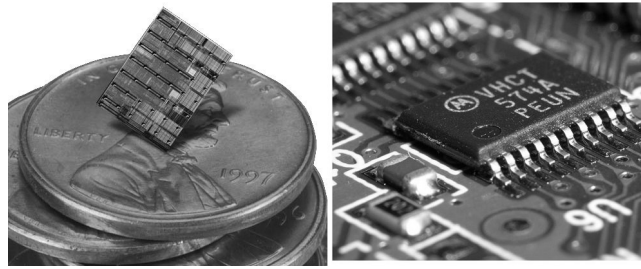



Figura 1.20: Comparação do tamanho do circuito integrado com uma moeda (esquerda) e um chip (direita).

Mas o diferencial dos circuitos integrados não era o apenas o tamanho, mas o processo de fabricação que possibilitava a construção de vários circuitos simultaneamente, facilitando a produção em massa. Este avanço pode ser comparado ao advento da impressa, que revolucionou a produção dos livros.

Nota

Didaticamente os circuitos integrados são categorizados de acordo com a quantidade de integração que eles possuem:

- LSI (Large Scale Integration - 100 transistores): computadores da terceira geração
- VLSI (Very Large Scale Integration - 1.000 transistores): computadores da quarta geração
- ULSI (Ultra-Large Scale Integration - milhões de transistores): computadores da quinta geração

Um computador que representa esta geração foi o *IBM's System/360*, voltado para o setor comercial e científico. Ele possuía uma arquitetura plugável, na qual o cliente poderia substituir as peças que dessem defeitos. Além disso, um conjunto de periféricos eram vendidos conforme a necessidade do cliente.

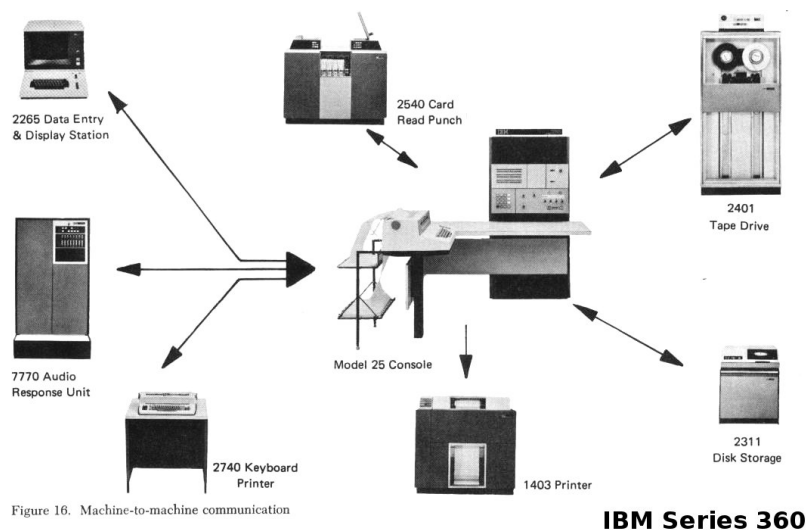



Figura 1.21: Arquitetura plugável da série 360 da IBM.

A IBM, que até então liderava o mercado de computadores, passou a perder espaço quando concorrentes passaram a vender periféricos mais baratos e que eram compatíveis com sua arquitetura. No final desta geração já começaram a surgir os computadores pessoais (Figura 1.22 [15]).

Figura 1.22: Computador Apple I.

Outro evento importante desta época foi que a IBM passou a separar a criação de hardware do desenvolvimento de sistemas, iniciando o mercado da indústria de softwares. Isto foi possível devido a utilização das linguagens de alto nível nestes computadores.

Linguagem de alto nível

Uma linguagem é considerada de alto nível quando ela pode representar ideias abstratas de forma simples, diferente da linguagem de baixo nível que representa as próprias instruções de máquina.

Exemplo de linguagem de alto nível:

```
x = y*7 + 2
```

Mesmo código em baixo nível (assembly):

```
load y    // carrega valor de y
mul 7     // multiplica valor carregado por 7
add 2     // adiciona 2
store x   // salva o valor do último resultado em x
```

Os códigos `load`, `mul`, `add` e `store` são os *mnemônicos* que representam as instruções em código de máquina (binário).

1.2.4 Quarta Geração (1977-1991)

Os computadores da quarta geração são reconhecidos pelo surgimento dos processadores — unidade central de processamento. Os sistemas operacionais como MS-DOS, UNIX, Apple's Macintosh foram construídos. Linguagens de programação orientadas a objeto como C++ e Smalltalk foram desenvolvidas. Discos rígidos eram utilizados como memória secundária. Impressoras matriciais, e os teclados com os layouts atuais foram criados nesta época.

Os computadores eram mais confiáveis, mais rápidos, menores e com maior capacidade de armazenamento. Esta geração é marcada pela venda de computadores pessoais (Figura 1.23 [16]).

Figura 1.23: Computador pessoal da quarta geração.

1.2.5 Quinta Geração (1991 — dias atuais)

Os computadores da quinta geração usam processadores com milhões de transistores. Nesta geração surgiram as arquiteturas de 64 bits, os processadores que utilizam tecnologias RISC e CISC, discos rígidos com capacidade superior a 600GB, pen-drives com mais de 1GB de memória e utilização de disco ótico com mais de 50GB de armazenamento.

Figura 1.24: Computador da quinta geração.

A quinta geração está sendo marcada pela **inteligência artificial** e por sua **conectividade**. A inteligência artificial pode ser verificada em jogos e robores ao conseguir desafiar a inteligência humana. A conectividade é cada vez mais um requisito das indústrias de computadores. Hoje em dia, queremos que nossos computadores se conectem ao celular, a televisão e a muitos outros dispositivos como geladeira e câmeras de segurança.

1.3 Recapitulando

Estudamos neste capítulo a história do computador. Conhecemos os precursores do computador, iniciando pelo o ábaco que auxiliava a humanidade na realização de cálculos. Muitos séculos depois, Napier descobriu os logaritmos e inventou os ossos de Napier. Pascal inventou uma máquina que era capaz de realizar somas e subtrações através de engrenagens.

Também vimos que no século XIX, o Tear de Jacquard introduziu o uso de cartões perfurados, e mais tarde, Charles Babbage adaptou a ideia para o uso em sistemas computacionais, embora nunca tenha terminado a construção de suas máquinas.

Em seguida, concluímos os estudos do capítulo aprendendo sobre as gerações dos computadores, inicialmente com o uso de velas, depois com transistores e finalmente com a utilização de circuitos integrados e como eles revolucionaram a fabricação dos computadores.

Capítulo 2

Representação da informação

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Explicar o que são bit e byte e a importância do último para representação da informação
- Explicar como números, textos, imagens e músicas podem ser expressos através de sequências de bits
- Ser capaz de representar novas informações através de bits, com o auxílio de uma tabela

Você provavelmente já ouviu alguém falar que os computadores trabalham internamente apenas com **0** e **1** (zero e um). Tudo o que você assiste, escuta ou cria no computador, é processado internamente através de sequências de zeros e uns. O computador ao ler estas sequências, consegue interpretá-las e em seguida apresentar as informações contidas nelas.

Neste capítulo estudaremos o conceito e bit e byte e como eles podem ser utilizados para representar diversas informações.

Vamos começar nossos estudos aprendendo sobre os **bits** e **bytes**.

2.1 Conceito de bit e byte

Um **bit** ou dígito binário (*binary digit*), é a unidade básica que os computadores e sistemas digitais utilizam para trabalhar, ele pode assumir apenas dois valores, **0** ou **1**. Um **byte** é uma sequência de 8 bits.

Fisicamente, um bit pode ser representado de várias formas: através de dois valores de voltagem aplicados num fio, diferentes direções de magnetização em uma fita magnética, entre outras. O importante é que seja possível identificar dois estados diferentes.

Importante
O byte é a menor unidade de armazenamento utilizada pelos computadores. Isto quer dizer que, nós nunca conseguiremos salvar menos do que 8 bits.

Na próxima seção iremos estudar como os bits e bytes são utilizados na representação de dados e mídias.

2.2 Possibilidades de representação

Como um bit só pode assumir dois valores (**0** ou **1**), só será possível representar exatamente dois estados distintos. Na Tabela 2.1 [18] nós temos exemplos de como podemos associar significados aos valores do bit.

Por exemplo, em um sistema com trava eletrônica, o valor **0** poderia indicar que a porta estava fechada, enquanto **1** indicaria que a porta está aberta. Em outro sistema que registra o estado civil, **0** poderia representar *Solteiro*, enquanto **1** seria *Casado*.

Tabela 2.1: Representações com um bit.

Bit	Porta	Lâmpada	Sexo	Detector de movimento	Estado civil
0	Fechada	Desligada	Masculino	Sem movimento	Solteiro
1	Aberta	Ligada	Feminino	Com movimente	Casado

Para representar mais de dois valores distintos nós precisamos de uma sequência de bits maior. Na Tabela 2.2 [18] nós temos exemplos de representações utilizando sequências com **2** bits, obtendo **4** possibilidades. Nesta caso, o estado civil *Casado* passou a ser representado pela sequência `01`.

Tabela 2.2: Representações com dois bits.

Sequencia de Bits	Lâmpada	Estado civil
00	Desligada	Solteiro
01	Ligada com intensidade **baixa**	Casado
10	Ligada com intensidade **alta**	Divorciado
11	*Não utilizado*	Viúvo

Nota

Observe que o número de possibilidades diferentes que podemos representar depende do tamanho da sequência de bits que estamos utilizando, mais precisamente: $2^{tamanho}$.

2^1 = 2	2^2 = 4	2^3 = 8	2^4 = 16
2^5 = 32	2^6 = 64	2^7 = 128	2^8 = 256 possibilidades (um byte)
16 bits = 65.535	32 bits = 4.294.967.295		64 bits = 18.446.744.073.709.551.615

As tabelas são bastante utilizadas para representar informações. Em uma coluna colocamos o que desejamos representar e na outra sua representação binária. Não há uma ordem particular, por exemplo, na Tabela 2.2 [18] *Solteiro* era representado por `00`, mas poderíamos construir outra tabela em que seria **codificado** como `11` (lê-se um-um).

Importante
O termo **codificar** significa traduzir um conteúdo para a sua representação binária.

Percebam também que, quando todas as informações desejadas já foram representadas, podem existir sequências binárias que não possuem significado definido, que foi o caso da sequência `11` para a lâmpada (Tabela 2.2 [18]).

2.2.1 Números

Independente do que desejamos representar, o primeiro passo é verificar quantas informações diferentes iremos utilizar e, com base nestas informações podemos calcular quantos bits serão necessários para representar todas as possibilidades.

Para representar números é necessário estabelecer o intervalo que desejamos utilizar, pois precisamos definir quantas possibilidades diferentes queremos representar. Já vimos que com 8 bits podemos representar 256 possibilidades (números) diferentes. Para representar números inteiros[1] e positivos podemos construir uma tabela com todas estas possibilidades. Na Tabela 2.3 [19] temos exemplos de como alguns desses números são representados.

Tabela 2.3: Representação de números utilizando um byte.

Num	Byte	Num	Byte	Num	Byte	Num	Byte	Num	Byte
0	00000000	8	00001000	16	00010000	24	00011000	248	11111000
1	00000001	9	00001001	17	00010001	25	00011001	249	11111001
2	00000010	10	00001010	18	00010010	26	00011010	250	11111010
3	00000011	11	00001011	19	00010011	27	00011011	251	11111011
4	00000100	12	00001100	20	00010100	28	00011100	252	11111100
5	00000101	13	00001101	21	00010101	29	00011101	253	11111101
6	00000110	14	00001110	22	00010110	30	00011110	254	11111110
7	00000111	15	00001111	23	00010111	31	00011111	255	11111111

Nota
Usualmente os bits são representados utilizando o sistema **hexadecimal**, pois eles ocupam menos espaço. Por exemplo, a sequência de bits `1111 1000` é expressa por `F8` em hexadecimal.

2.2.2 Texto

Nesta seção nós iremos aprender como o computador representa texto. Novamente, podemos utilizar uma tabela definindo os caracteres que desejamos representar e suas correspondências binárias.

O ASCII é o padrão de representação de caracteres mais conhecido. Na Tabela 2.4 [20] é apresentado um extrato da tabela ASCII, onde cada caractere possui sua representação em bits. Este padrão

[1] As representações de números negativos e reais (ponto flutuante) também são possíveis utilizando outras estratégias.

também inclui outros caracteres de controle, não apresentados na tabela, como *fim de linha* e *final de arquivo*. A composição de um texto é realizada informado a sequência de caracteres contidos nele.

Tabela 2.4: Extrato da tabela ASCII

Caractere	Byte	Caractere	Byte	Caractere	Byte	Caractere	Byte	Caractere	Byte
a	01100001	A	01000001	n	01101110	N	01001110	0	00110000
b	01100010	B	01000010	o	01101111	O	01001111	1	00110001
c	01100011	C	01000011	p	01110000	P	01010000	2	00110010
d	01100100	D	01000100	q	01110001	Q	01010001	3	00110011
e	01100101	E	01000101	r	01110010	R	01010010	4	00110100
f	01100110	F	01100110	s	01110011	S	01010011	5	00110101
g	01100111	G	01100111	t	01110100	T	01010100	6	00110110
h	01101000	H	01101000	u	01110101	U	01010101	7	00110111
i	01101001	I	01101001	v	01110110	V	01010110	8	00111000
j	01101010	J	01101010	w	01110111	W	01010111	9	00111001
k	01101011	K	01101011	x	01111000	X	01011000		
l	01101100	L	01001100	y	01111001	Y	01011001		
m	01101101	M	01001101	z	01111010	Z	01011010		

Importante

Percebam que neste sistema os caracteres são representados por exatamente um byte, que é o tamanho mínimo possível de ser salvo no computador.

Talvez você tenha percebido a ausência dos *caracteres especiais*, como o "ç", "ß", além dos caracteres acentuados como "ã", "ô", "é", etc. Isto porque o padrão ASCII foi criado por americanos para **codificar** as mensagens escritas no idioma inglês, que não possuem tais caracteres. Por esta razão, existem vários outros sistemas de codificação para melhor representar as mensagens do idioma que se deseja utilizar, alguns exemplos são: **Unicode**, **UTF-8** e **ISO 8859-1** (padrão latino-americano).

Dica

Faça um teste! Abra um editor de texto como o *bloco de notas*, *gedit* ou *kate* (não use o *Word*). Digite *abc* no documento em branco e salve-o. Em seguida, verifique o tamanho do arquivo, dependendo da codificação utilizada pelo seu editor o arquivo poderá ter de 3 a 8 bytes.

2.2.3 Imagem

Uma das formas possíveis para representar imagens é tratá-las como grades de pontos (ou *pixels*).

Ao atribuir uma cor para cada ponto, podemos então pintar a imagem. Na Figura 2.1 [21] nós temos uma imagem e um recorte em destaque, mostrando a grade de pontos com suas respectivas cores.

Além das cores dos pontos também é necessário definir o tamanho da grade (quantos pontos teremos na horizontal e na vertical), também conhecida como *resolução da imagem*. Sem a resolução teríamos apenas um linha de pontos coloridos.

Figura 2.1: Fotografia evidenciando a grade de pontos.

Um sistema popular de representação de cores é o **RGB**, onde é reservado um byte para os tons de cada uma das cores primárias: vermelho, verde e azul. Como um byte permite representar 256 tons de uma cor, ao total são possíveis representar 16 milhões (256x256x256) de cores.

Nota

Através do sistema **RGB** podemos representar as três cores primárias e as suas derivadas, que são as cores resultantes das misturas das cores primárias. Neste sistema, o branco é interpretado como sendo a união de todas as cores, e o preto a ausência de cor. Este sistema é utilizado pelo formato de imagem **BMP**.

O sistema **RGBA** inclui também um *canal alpha*, responsável por representar a transparência do ponto, utilizado pelo formato de imagem **PNG**.

2.2.4 Música

Para representar uma música, podemos imaginá-la como sendo apenas uma partitura e salvar todas as informações contidas nela. Depois a música poderá ser ouvida tocando a partitura salva.

Figura 2.2: Partitura da música Marcha Soldado.

Nota

Os toques dos antigos celulares monofônicos utilizavam este sistema para reproduzir as músicas, salvando apenas as notas dos tons. O formato de arquivo **MID** utiliza esta estratégia. Você pode baixar esta música em: https://github.com/edusantana/introducao-a-computacao-livro/raw/master/arquivos/midias/marcha_soldado.mid

Você deve está pensando, "Mas as músicas **MP3** que escuto também tem voz, como ela é representada?". Os sons também podem ser representados através das frequências de ondas (Figura 2.3 [22]) que os caracterizam. Mais tarde, quando você desejar escutar as músicas, o computador será capaz de reproduzir os mesmos sons.

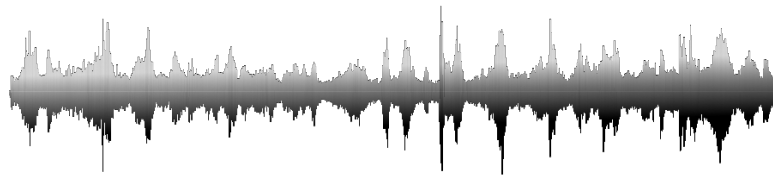



Figura 2.3: Representação de uma música através de ondas.

Se possível, acessem este vídeo sobre a representação da informação.

Figura 2.4: Vídeo sobre Representação da Informação: http://youtu.be/y_eCItEibHI

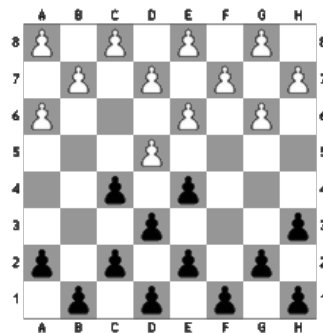
2.3 Recapitulando

Neste capítulo você aprendeu o que são bits e bytes, e a importância do último para salvar dados no computador.

Em seguida, vimos como dados podem ser representados através de uma tabela, e como calcular quantos bits são necessários para representar as informações que desejamos.

Por último, vimos como números, textos, imagens e músicas podem ser representadas através de bits.

2.4 Atividades

1. Utilizando o método de representação de números estudado (Seção 2.2.1 [19]), represente os números **32**, **64** e **128** em bytes.
2. Como você representaria todos os meses do ano em bits?
3. Quantos bits são necessários para representar 150 possibilidades diferentes?
4. ★ Como você representaria as horas em bits?
5. ★ Como você representaria o tabuleiro abaixo em binário? Quantos bytes sua estratégia utiliza?

Capítulo 3

Sistemas de numeração

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Entender a origem dos sistemas de numeração na história
- Explicar o funcionamento do sistema de numeração binário
- Ser capaz de efetuar operações da aritmética e lógica binária
- Compreender como a lógica binária permite criar operações mais complexas no computador

Um numeral é um símbolo ou grupo de símbolos que representa um número em um determinado instante da história humana. Tem-se que, numa determinada escrita ou época, os numerais diferenciaram-se dos números do mesmo modo que as palavras se diferenciaram das coisas a que se referem. Os símbolos "11", "onze"e "XI"(onze em latim) são numerais diferentes, representativos do mesmo número, apenas escrito em idiomas e épocas diferentes. Este capítulo debruça-se sobre os vários aspectos dos sistemas de numerais.

Entraremos em mais detalhes sobre o sistema de numeração binária devido a sua utilização nos computadores, permitindo assim, que o mesmo realize pequenas operações aritméticas que servirão como base para grandes operações como busca, ordenação e indexação de informação entre outras operações comuns do dia a dia.

3.1 Noções de Sistema de Numeração

Há milhares de anos o modo de vida era muito diferente do atual. Os homens primitivos não tinham necessidade de contar. Eles não compravam, não vendiam, portanto não usavam dinheiro.

Com o passar dos anos, os costumes foram mudando e o homem passou a cultivar a terra, a criar animais, a construir casas e a comercializar. Com isso, surgiu a necessidade de contar.

A vida foi tornando-se cada vez mais complexa. Surgiram as primeiras aldeias que, lentamente, foram crescendo, tornando-se cidades. Algumas cidades se desenvolveram, dando origem às grandes civilizações. Com o progresso e o alto grau de organização das antigas civilizações, a necessidade de aprimorar os processos de contagem e seus registros tornou-se fundamental.

Foram criados, então, símbolos e regras originando assim os diferentes sistemas de numeração.

3.1.1 Sistema de numeração Egípcio (3000 a.C.)

Um dos primeiros sistemas de numeração que temos conhecimento é o egípcio, quefoi desenvolvido pelas civilizações que viviam no vale do Rio Nilo, ao nordeste da África.

Observem, na Figura 3.1 [25], os símbolos e a representação de alguns números nesse sistema de numeração.

Símbolo	Nome	Valor
	Bastão	1
	Calcanhar	10
	Rolo de corda	100
	Flor de lótus	1000
	Dedo apontado	10000
	Peixe	100000
	Homem	1000000

Figura 3.1: Sistema de Numeração Egípcio

Este sistema adota o princípio aditivo, ou seja, os símbolos possuem seus respectivos valores individuais e juntos passam a formar novos valores pela simples adição destes.

3.1.2 Sistemas de numeração Babilônico (2000 a.C.)

Os babilônios viviam na Mesopotâmia, nos vales dos rios Tigres e Eufrates, na Ásia. Esta região é ocupada atualmente pelo Iraque.

Na escrita dos números, o sistema de numeração dos babilônios se parecia muito com o sistema de numeração desenvolvido pelos egípcios, ambos eram aditivos. Observe, na Figura 3.2 [26], os símbolos e a representação de alguns números, de 7 a 59, nesse sistema de numeração.

Figura 3.2: Sistema de Numeração Babilônico

Dica

Agora é com você. Qual seria o valor que cada símbolo babilônico, seguindo os exemplos da Figura 3.2 [26]?

= ?

= ?

3.1.3 Sistema de numeração Romano

O sistema de numeração romano, apesar das dificuldades operatórias que apresentava, foi utilizado na Europa durante muitos séculos. Esse sistema de numeração foi desenvolvido pela civilização romana, cuja sede era a cidade de Roma, situada na Itália.

Ainda hoje, utilizamos esse sistema de numeração em algumas situações, tais como:

- na designação de papas e reis;
- na designação de séculos e datas;
- na indicação de capítulos e volumes de livros;
- nos mostradores de alguns relógios, etc.

Com o passar dos anos, o sistema de numeração romano (Figura 3.3 [27]) sofreu um longo processo de evolução. Inicialmente, os romanos utilizavam apenas o principio aditivo, sendo que um mesmo símbolo podia ser repetido até, no máximo, quatro vezes. Posteriormente, eles evoluíram este sistema, passando a utilizar também o princípio subtrativo, além de permitir a repetição de um mesmo símbolo, no máximo, três vezes.

Numeração romana antiga Princípio aditivo	Numeração romana moderna Princípio subtrativo
I I I I 1 + 1 + 1 + 1 = 4	I V 5 – 1 = 4
V I I I I 5 + 1 + 1 + 1 + 1 = 9	I X 10 -1 = 9

	Princípio aditivo	Princípio subtrativo e aditivo
Nossa numeração	Numeração romana antiga	Numeração romana moderna
1	I	I
2	II	II
3	III	III
4	IIII	IV
5	V	V
6	VI	VI
7	VII	VII
8	VIII	VIII
9	VIIII	IX
10	X	X
11	XI	XI
12	XII	XII

	Princípio aditivo	Princípio subtrativo e aditivo
Nossa numeração	Numeração romana antiga	Numeração romana moderna
16	XVI	XVI
17	XVII	XVII
18	XVIII	XVIII
19	XVIIII	XIX
20	XX	XX
40	XXXX	XL
50	L	L
90	LXXXX	XC
100	C	C
400	CCCC	CD
500	D	D
1000	M	M

Figura 3.3: Sistema de Numeração Romano

3.1.4 Sistemas de numeração Indo-Arábico

Os hindus, que viviam no vale do Rio Indo, onde hoje é o Paquistão, conseguiram desenvolver um sistema de numeração que reunia as diferentes características dos antigos sistemas.

Tratava-se de um sistema posicional decimal. Posicional porque um mesmo símbolo representava valores diferentes dependendo da posição ocupada, e decimal porque era utilizado um agrupamento de dez símbolos.

Esse sistema posicional decimal, criado pelos hindus, corresponde ao nosso atual sistema de numeração, já estudado por você nas séries anteriores. Por terem sido os árabes os responsáveis pela divulgação desse sistema. Ele ficou conhecido como sistema de numeração indo-arábico. Os dez símbolos, utilizados para representar os números, denominam-se algarismos indo-arábicos. São eles:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Dica

Veja, na Figura 3.4 [28][a], as principais mudanças ocorridas nos símbolos indo-arábicos, ao longo do tempo.

Figura 3.4: Sistema de numeração Indo-Arábico

Observe que, inicialmente, os hindus não utilizavam o zero. A criação de um símbolo para o **nada**, ou seja, o zero, foi uma das grandes invenções dos hindus.

[a] Fonte: http://www.programandoomundo.com/Sistema%20Indo-Arabico.htm.

3.2 Sistema de Numeração Posicional

O método de numeração de quantidades ao qual estamos acostumados, utiliza um sistema de numeração posicional. Isto significa que a posição ocupada por cada algarismo em um número altera seu valor de uma potência de 10 (na base 10) para cada casa à esquerda.

Por exemplo:

No sistema decimal (base 10), no número 125 o algarismo 1 representa 100 (uma centena ou 10^2), o 2 representa 20 (duas dezenas ou 2 x 10^1), o 5 representa 5 mesmo (5 unidades ou 5x10^0).

Assim, em nossa notação:

$$125 = 1\times10^2 + 2\times10^1 + 5\times10^0$$

3.2.1 Base de um Sistema de Numeração

A base de um sistema é a quantidade de algarismos disponíveis na representação. A base 10 é hoje a mais usualmente empregada, embora não seja a única utilizada. No comércio pedimos uma dúzia de rosas (base 12) e marcamos o tempo em minutos e segundos (base 60).

Quando lidamos com computadores, é muito comum e conveniente o uso de outras bases, as mais importantes são a binária (base 2), octal (base 8) e hexadecimal (base 16).

Um sistema numérico de base k precisa de k símbolos diferentes para representar seus dígitos de 0 a k-1. Os números decimais são formados a partir de 10 dígitos decimais:

```
0 1 2 3 4 5 6 7 8 9
```

Já os números na base binária são representados a partir de dois dígitos:

```
0 1
```

O octal necessita de oito:

```
0 1 2 3 4 5 6 7
```

No caso de números hexadecimais, são necessários 16 algarismos. Portanto, serão mais 6 símbolos além dos dez algarismos arábicos. Em geral, usam-se as letras maiúsculas de A a F:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

A representação 318_{10} (base 10), significa:

$$318_{10} = 3\times10^2 + 1\times10^1 + 8\times10^0$$

Generalizando, representamos uma quantidade N qualquer, numa dada base b, com um número tal como segue:

$$N_b = a_0 \times b^n + a_1 \times b^{n-1} + \ldots + a_n \times b^0$$

Abaixo, o número 35 será expresso nas bases elencadas acima:

Decimal
$35 = 3\times10^1 + 5\times10^0 = 30 + 5 = 35_{10}$

Binário
$35 = 1\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 32 + 0 + 0 + 0 + 2 + 1 = 100011_2$

Octal
$35 = 4\times8^1 + 3\times8^0 = 32 + 3 = 43_8$

Hexadecimal
$35 = 2\times16^1 + 3\times16^0 = 32 + 3 = 23_{16}$

3.3 Conversões entre bases

Nesta seção iremos analisar as regras gerais para converter números entre duas bases quaisquer.

3.3.1 Conversões entre as bases 2, 8 e 16

As conversões mais simples são as que envolvem bases que são potências entre si. Vamos exemplificar com a conversão entre a base 2 e a base 8. Como $2_3 = 8$, então a conversão funciona da seguinte forma:

separando os algarismos de um número binário (base 2) em grupos de três algarismos (começando sempre da direita para a esquerda) e convertendo cada grupo de três algarismos para seu equivalente em octal, teremos a representação do número em octal.

Por exemplo:

$10101001_2 = 010 . 101 . 001_2$

Olhando a tabela de conversão direta temos:

Tabela 3.1: Conversão direta de binário para octal e vice-versa

Binário	**Octal**
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Logo:

$010_2 = 2_8$	$101_2 = 5_8$	$001_2 = 1_8$	$10101001_2 = 251_8$

Vamos agora exemplificar com uma conversão entre as bases 2 e 16. Como $2^4 = 16$, seguindo o processo anterior, basta separarmos em grupos de quatro algarismos e converter cada grupo seguindo a Tabela 3.2 [30].

Por exemplo:

$11010101101_2 = 0110 . 1010 . 1101_2$

Olhando a tabela de conversão direta temos:

Tabela 3.2: Conversão direta de binário para hexadecimal e vice-versa.

Binário	**Hexadecimal**	**Binário**	**Hexadecimal**
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D

Tabela 3.2: (continued)

Binário	Hexadecimal	Binário	Hexadecimal
0110	6	1110	E
0111	7	1111	F

Logo:

$0110_2 = 6_{16}$	$1010_2 = A_{16}$	$1101_2 = D_{16}$	$11010101101_2 = 6AD_{16}$

Vamos agora analisar a conversão inversa.

Por exemplo:

$$A81_{16} = A\ .\ 8\ .\ 1_{16}$$

Sabemos que:

$A_{16} = 1010_2$	$8_{16} = 1000_2$	$1_{16} = 0001_2$

Portanto:

$$A81_{16} = 101010000001_2$$

3.3.2 Conversão de números em uma base b qualquer para a base 10.

Vamos lembrar a expressão geral já apresentada:

$$N_b = a_0 \times b^n + a_1 \times b^{n-1} + \ldots + a_n \times b^0$$

A melhor forma de fazer a conversão é usando essa expressão. Como exemplo, o número 101101_2 terá calculado seu valor na base 10:

$$101101_2 = 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 45_{10}$$

Outros exemplos:

Converter $A5_{16}$ para a base 10:

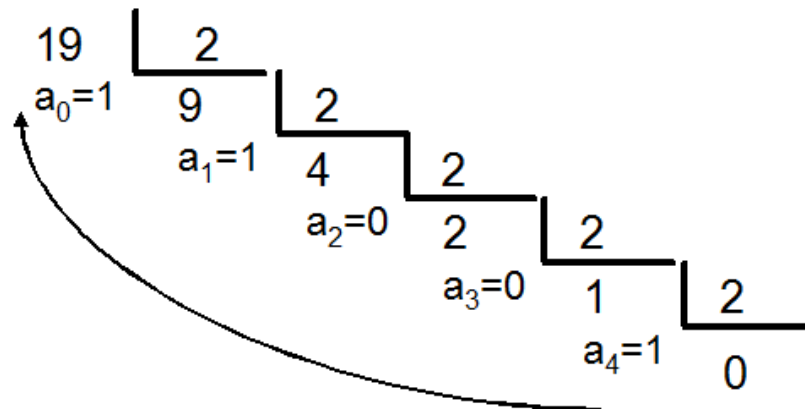
$$A5_{16} = 10\times16^1 + 5\times16^0 = 160 + 5 = 165_{10}$$

Converter 485_9 para a base 10:

$$485_9 = 4\times9^2 + 8\times9^1 + 5\times9^0 = 324 + 72 + 5 = 401_{10}$$

3.3.3 Conversão de números da base 10 para uma base b qualquer

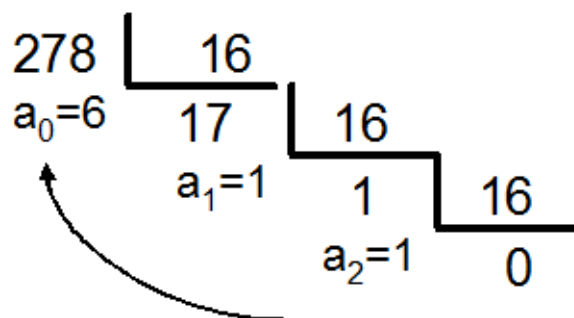
A conversão de números da base 10 para uma base qualquer, emprega algoritmos que serão o inverso dos anteriores. O número decimal será dividido sucessivas vezes pela base, o resto de cada divisão ocupará sucessivamente as posições de ordem 0, 1, 2 e assim por diante, até que o resto da última divisão (que resulta em quociente 0) ocupe a posição de mais alta ordem.

- Conversão do número 19_{10} para a base 2:

Logo temos:

$19_{10} = 10011_2$

Usando a conversão anterior como prova real, temos:

$10011_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 19_{10}$

Conversão do número 278_{10} para a base 16:

Logo temos:

$278_{10} = 116278_{16}$

3.4 Números Binários Negativos

Os computadores lidam com números positivos e números negativos, sendo necessário encontrar uma representação para números com sinal negativo. Existe uma grande variedade de opções, das quais nesta seção serão apresentadas apenas três para representar valores negativos:

- sinal e amplitude/magnitude (S+M)
- complemento de 1
- complemento de 2

3.4.1 Sinal e Amplitude/Magnitude (S + M)

Como o próprio nome indica, a representação **sinal** e **amplitude** utiliza um bit para representar o sinal, o bit mais à esquerda: **0** para indicar um valor positivo, **1** para indicar um valor negativo.

$$+10_{10} = \mathbf{0}1010_2 \qquad -10_{10} = \mathbf{1}1010_2$$

3.4.2 Complemento de 1

Na representação em complemento de 1 invertem-se todos os bits de um número para representar o seu complementar: assim, se converte um valor positivo para um negativo, e vice-versa. Quando o bit mais à esquerda é 0, esse valor é positivo; se for 1, então é negativo.

Exemplo:

$100_{10} = 01100100_2$ (com 8 bits)

Invertendo todos os bits:

$10011011_2 = -100_{10}$

Importante

O problema desta representação é que existem 2 padrões de bits para o 0, havendo assim desperdício de representação:

$0_{10} = 00000000_2 = 11111111_2$

3.4.3 Complemento de 2

A solução encontrada consiste em representar os números em **complemento de 2**. Para determinar o negativo de um número, inverte-se todos os seus bits e soma-se uma unidade.

Exemplo:

Representação binária

$101_{10} = 01100101_2$ (com 8 bits)

Invertendo todos os bits

10011010_2

Somando uma unidade

$10011010_2 + 1 = 10011011_2 = -101_{10}$

A representação em complemento para 2 tem as seguintes características:

- o bit da esquerda indica o sinal;
- possui processo para converter um número de positivo para negativo e de negativo para positivo;
- o 0 tem uma representação única: todos os bits a 0;
- a gama de valores que é possível representar com n bits é $-2^{n-1} \ldots 2^{n-1}-1$.

Exemplo:

Qual o número representado por 11100100_2 (com 8 bits)? Como o bit da esquerda é 1 este número é negativo. Invertendo todos os bits:

$$00011011_2$$

Somando uma unidade:

$$00011011_2 + 1 = 00011100_2 = 28_{10}$$

Logo:

$$11100100_2 = -28_{10}$$

3.5 Aritmética Binária

Como o computador manipula os dados (números) através de uma representação binária, iremos estudar agora a aritmética do sistema binário, a mesma usada pela ULA (Unidade Lógica e Aritmética) dos processadores.

3.5.1 Soma e Subtração Binária

A tabuada da soma aritmética em:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (e "vai um" para o dígito de ordem superior)
1 + 1 + 1 = 1 (e "vai um" para o dígito de ordem superior)
```

Por exemplo:

Efetuar $011100_2 + 011010_2$

Nota
Soma-se as posições da direita para esquerda, tal como uma soma decimal.

Solução:

```
   0¹ 1¹ 1 1 0 0    ← "vai um"
 + 0  1  1 0 1 0
   1  1  0 1 1 0
```

A tabuada da subtração aritmética binária:

```
0 - 0 = 0
0 - 1 = 1 ("vem um do próximo")
1 - 0 = 1
1 - 1 = 0
```

Nota
Como é impossível tirar 1 de 0, o artifício é "pedir emprestado" 1 da casa de ordem superior, ou seja, na realidade o que se faz é subtrair 1_2 de 10_2 e encontramos 1_2 como resultado, devendo então subtrair 1 do dígito de ordem superior. Este algoritmo é exatamente o mesmo da subtração em decimal.

Por exemplo: $111100_2 - 011010_2 = ?$

Nota
não esqueça, subtrai-se as colunas da direita para a esquerda, tal como uma subtração decimal.

Solução:

```
   1 1⁰1²0 0
 - 0 1 0 1 0
   1 0 0 1 0
```

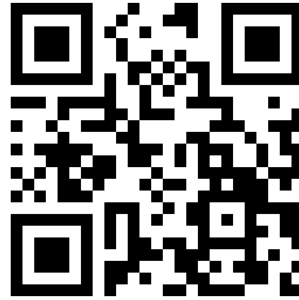



Figura 3.5: Vídeo sobre Soma e Subtração Binária: http://youtu.be/NeQBC9Z5FHk

3.5.2 Subtração nos computadores

Na eletrônica digital de dispositivos tais como computadores, circuitos simples custam menos e operam mais rápido do que circuitos mais complexos. Logo, números em complemento de dois são usados na aritmética, pois eles permitem o uso dos circuitos mais simples, baratos e rápidos.

Uma característica do sistema de complemento de dois é que tanto os números com sinal quanto os números sem sinal podem ser somados pelo mesmo circuito. Por exemplo, suponha que você deseja somar os números sem sinal 132_{10} e 14_{10}.

10000100_2	$+132_{10}$
$+\underline{00001110_2}$	$+14_{10}$
10010010_2	$+146_{10}$

O microprocessador tem um circuito na ULA (Unidade Lógica e Aritmética) que pode somar números binários sem sinal, quando aparece o padrão 10000100_2 em uma entrada e 00001110_2 na outra entrada, resulta 10010010_2 na saída.

Surge a pergunta: como a ULA sabe que os padrões de bits nas entradas representam número sem sinal e não números em complemento de dois? E a resposta é: não sabe. A ULA sempre soma como se as entradas fossem números binários sem sinal. Sempre produzirá o resultado correto, mesmo se as entradas forem números em complemento de dois.

10000100_2	-124_{10}
$+\underline{00001110_2}$	$+14_{10}$
10010010_2	-110_{10}

Isto comprova um ponto muito importante. O somador na ULA sempre soma padrões de bits como se eles fossem números binários sem sinal. É a nossa interpretação destes padrões que decide se números com ou sem sinal estão sendo tratados. O bom do complemento de dois é que os padrões de bits podem ser interpretados de qualquer maneira. Isto nos permite trabalhar com números com e sem sinal sem requerer diferentes circuitos para cada padrão.

A aritmética de complemento de dois também simplifica a ULA em outro ponto. Todo microprocessador precisa da instrução de subtração. Assim, a ULA deve ser capacitada a subtrair um número de outro. Entretanto, se isto necessitar de um circuito de subtração separado, a complexidade e o custo da ULA seriam aumentados. Felizmente, a aritmética de complemento de dois permite a ULA, realizar operações de subtração usando um circuito somador. Ou seja, a CPU usa o mesmo circuito tanto para soma como para subtração.

3.5.2.1 Subtração em complemento de dois

Uma vez que o complemento de dois foi formado, a CPU pode realizar uma subtração indiretamente pela adição do complemento de dois do Subtraendo com Minuendo. Não esquecendo de ignorar o último transporte da adição.

Como exemplo temos a subtração de 69_{10} (Minuendo) por 26_{10} (Subtraendo).

Jogue fora o transporte final:

Minuendo	0	1	0	0	0	1	0	1	69	
Complemento de dois do Subtraendo	1	1	1	0	0	1	1	0	74	complemento de dez
Diferença	0	0	1	0	1	0	1	1	43	

Fica o desafio de descobrir porque o valor 74_{10} é o complemento de 10 do número 26_{10}, a regra é análoga do complemento de 2 binária, ou seja, primeiro deve ser feito o complemento de 9 para cada número individualmente e depois deve ser somado o valor 1.

Este método permite à CPU realizar subtração e adição com o mesmo circuito. O método que a CPU usa para realizar subtração é de pouca importância para o uso de microprocessadores.

3.5.3 Multiplicação e divisão binária

Vamos ver agora a tabuada da multiplicação:

```
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
```

Nota
O processo é idêntico à multiplicação entre números decimais.

Exemplo:

Efetuar: 101_2 x 110_2

Solução:

$$
\begin{array}{rcl}
101_2 & \rightarrow & 5_{10} \\
\times\ \underline{110_2} & \rightarrow & 6_{10} \\
000 & & \\
101 & & \\
\underline{101} & & \\
11110_2 & \rightarrow & 30_{10}
\end{array}
$$

No entanto, a multiplicação em computadores é feita, também, por um artifício: para multiplicar A por n somamos A com A (n-1) vezes.

Exemplo:

4 x 3 = 4 + 4 + 4 = 12

E a divisão também pode ser feita por subtrações sucessivas, até o resultado zerar ou ficar negativo. Este artifício serve apenas para divisões inteiras.

Por exemplo:

16 ÷ 4 → 16 **- 4** = 12 → 12 **- 4** = 8 → 8 **- 4** = 4 → 4 **– 4** = 0

O número de subtrações indica o resultado da divisão inteira, neste caso, igual a 4.

Nota

Podemos concluir que qualquer operação aritmética pode ser realizada em computadores através de somas (diretas ou em complemento). Com isso diminui-se o número de circuitos lógicos de operações para o processador.

3.6 Representação de Número Fracionário no Sistema Binário

3.6.1 Notação de Ponto Fixo

Esta notação conhecida como Notação de Ponto Fixo, utiliza um ponto que funciona da mesma forma que o ponto da notação decimal. Em outras palavras, os dígitos à esquerda do ponto representam a parte inteira do valor, funcionando da mesma forma que a notação binária. E os dígitos à direita do ponto representam a parte não inteira, sendo o expoente da base 2 decrementada em 1 a cada casa afastada do ponto.

Exemplo:

$10.101_2 = 1\times2^1 + 1\times2^0$ (parte inteira)
$+ 1\times2^{-1} + 0\times2^{-2} + 0\times2^{-3}$ (parte fracionária)
2 + 0,5 + 0,125 = 2,62510

3.6.1.1 Soma e subtração de números fracionários

Para somarmos e subtrairmos duas representações binárias contendo ponto, basta alinharmos os pontos e aplicarmos o mesmo algoritmo de adição ou subtração binária demonstrada anteriormente.

Exemplos:

$$
\begin{array}{rcl}
1\,0\,1_2 & \rightarrow & 5_{10} \\
\times\ \underline{1\,1\,0_2} & \rightarrow & 6_{10} \\
0\,0\,0 & & \\
1\,0\,1 & & \\
\underline{1\,0\,1} & & \\
1\,1\,1\,1\,0_2 & \rightarrow & 30_{10}
\end{array}
$$

3.7 Fundamentos da Notação de Ponto Flutuante

A utilização da notação de ponto flutuante é muito grande em computadores, tanto para cálculos matemáticos com precisão, renderização de imagens digitais (criação de imagens pelo computador) entre outros. Os processadores atuais se dedicam muito em aperfeiçoar a técnica de seus chips para a aritmética de ponto flutuante, devido à demanda de aplicativos gráficos e jogos 3D que se utilizam muito deste recurso.

Nesta subseção iremos descrever os fundamentos da aritmética de ponto flutuante, para isso, serão apresentados alguns conceitos básicos, que juntamente com os conceitos da seção anterior, servirão para o entendimento do processo desta aritmética em um sistema computacional.

3.7.1 Notação de Excesso

Para trabalhar com a Notação Ponto Flutuante, precisamos entender a representação dos números inteiros (negativos e não negativos) utilizando a Notação de Excesso.

Neste sistema, cada número é codificado como um padrão de bits, de comprimento convencionado. Para estabelecer um sistema de excesso, primeiro escolhemos o comprimento do padrão a ser empregado, em seguida, escrevemos todos os diferentes padrões de bits com este comprimento, na ordem em que eles seriam gerados se estivéssemos contando em binário.

Logo, observamos que o primeiro desses padrões, que representa um dígito 1 como seu bit mais significativo, figura aproximadamente no centro da lista.

Tabela 3.3: Notação de excesso com 3 bits.

Valor Binário (Notação de Excesso)	Valor Representado
000	-4
001	-3
010	-2
011	-1
100 (centro)	0
101	1
110	2
111	3

Como podemos observar na Tabela 3.3 [40], escolhemos este padrão para representar o ZERO, os padrões que o seguem serão utilizados para representar 1, 2, 3. . . , os padrões que o precedem serão adotados para a representação dos inteiros negativos -1, -2, -3. . .

Logo na Tabela 3.3 [40] podemos observar o código resultante para padrões de três bits de comprimento.

Exemplo:

O valor 1 equivale a 101_2	O valor -1 equivale a 011_2

Nota

No sistema de Notação de Excesso é fácil distinguir entre padrões que representam valores negativos e positivos, pois aqueles que apresentam um 0 no bit mais significativo são números negativos, servindo o mesmo como bit de sinal.

3.7.2 Notação de Ponto Flutuante

O primeiro ponto a ser discutido, é o motivo da criação da Notação de Ponto Flutuante, já que na seção anterior já tínhamos trabalhado com a representação de números não inteiros utilizando a Notação de Ponto Fixo.

O problema do Ponto Fixo, é que o tamanho da parte inteira e da fracionária fica fixo com relação a seu armazenamento em memória, logo para números com a parte apenas inteira, a região alocada para tratar a parte fracionária será inutilizada e vice-versa. Logo, para evitar este desperdício criou-se a Notação de Ponto Flutuante.

3.7.2.1 Ponto Flutuante

Vamos explicar a notação de ponto flutuante por meio de um exemplo que emprega somente um byte de memória.

Primeiramente, escolhemos o bit mais significativo do byte para ser o bit de sinal do número. Um '0' neste bit significa que o valor representado é não negativo, enquanto o '1' indica que é negativo.

Em seguida dividimos os sete bits restantes em dois grupos, o campo de expoente e o campo da mantissa, como mostrado na Figura 3.6 [41].

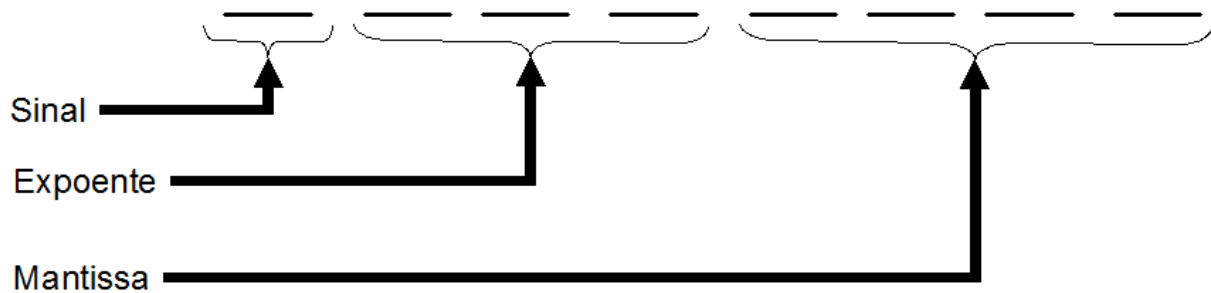



Figura 3.6: Divisão de 1 byte nos campos da Notação de Ponto Flutuante

Seja um byte contendo o padrão de bits 01101011. Interpretando este padrão no formato que acabamos de definir, constatamos que o bit de sinal é '0', o expoente é '110', e a mantissa é '1011'.

Para decodificarmos o byte, extraímos primeiro a mantissa e colocamos o ponto binário à sua esquerda, obtendo:

.1011

Em seguida, extraímos o conteúdo do campo do expoente e o interpretamos como um inteiro codificado em três bits pelo método de representação de excesso, nos dando o número positivo 2 (vide Tabela 3.3 [40]). Isto indica que devemos deslocar o ponto binário dois bits à direita (um expoente negativo codifica um deslocamento para a esquerda do ponto binário com a adição do valor 0).

Como resultado temos:

10.11

Que representa na Notação de Ponto Fixo:

$= 1\times2^{1} + 0\times2^{0} + 1\times2^{-1} + 1\times2^{-2}$
= 2+0+0,5+0,25 = 2,75

Em seguida, notamos que o bit de sinal do nosso exemplo é 0, assim, o valor representado é positivo (**+2,75**).

Figura 3.7: Vídeo sobre o Notação de Ponto Flutuante: http://youtu.be/psyH7eBVLr4

Nota
O uso da notação de excesso para representar o expoente no sistema de Ponto Flutuante se dá pela comparação relativa das amplitudes de dois valores, consistindo apenas em emparelhar as suas representações da esquerda para a direita em busca do primeiro bit em que os dois diferem. Assim, se ambos os bits de sinal forem iguais a '0', o maior dos dois valores é aquele que apresentar, da esquerda para a direita, um '1' no bit em que os padrões diferem, logo ao comparar:

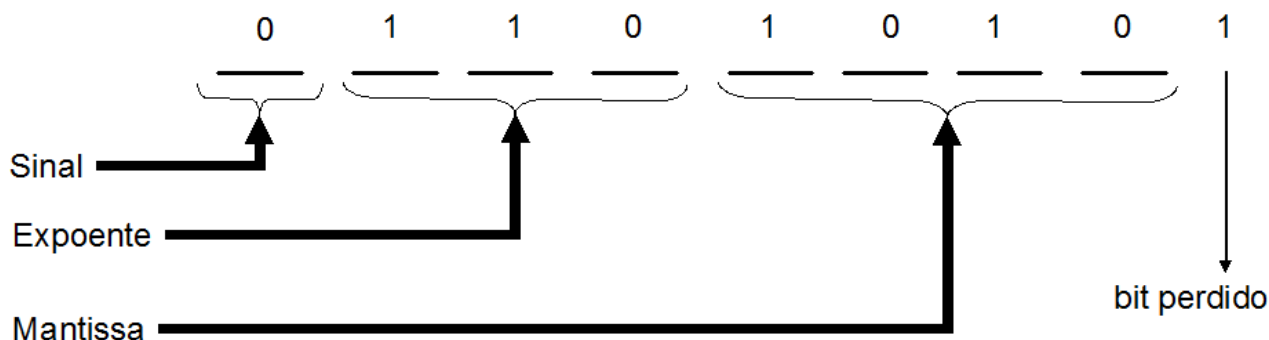
Conclui-se que o primeiro padrão é maior que o segundo, sem haver a necessidade de decodificar as representações em Ponto Flutuante, tarefa que seria mais custosa.

Quando os bits de sinal forem iguais a '1', o maior número é aquele que apresentar o '0' na diferença dos padrões (sempre percorrendo o número da esquerda para a direita).

3.7.3 Problema com Arredondamento

Consideremos o incômodo problema de representar o número 2,625 no sistema de ponto flutuante de um byte. Primeiramente, escrevemos 2,625 em binário, obtendo 10.101. Entretanto, ao copiarmos este código no campo da mantissa, não haverá espaço suficiente, e o último 1 (o qual representa a última parcela: 0,125) se perde, como pode ser observado na figura abaixo:

$2{,}625_{10} \rightarrow 10.101_2$
Expoente $\rightarrow$ 110
Mantissa $\rightarrow$ 10101

Logo temos:

Ao ignorarmos este problema, e continuarmos a preencher o campo do expoente e do bit do sinal, teremos o padrão de bits 01101010, que representa o valor 2,5 e não 2,625: o fenômeno assim observado é denominado erro de arredondamento, devido o campo da mantissa ter apenas quatro bits, enquanto, por questão de precisão, seriam necessários no mínimo cinco bits.

3.7.3.1 Overflow e Underflow

Os projetistas do hardware devem encontrar um compromisso entre a mantissa e o expoente dos números em ponto flutuante. A relação entre mantissa e expoente é expressa do seguinte modo: o aumento do número de bits reservados à mantissa aumenta a precisão do número, enquanto o aumento do número de bits reservados ao expoente aumenta o intervalo de variação dos números representados.

Quando o expoente é muito grande ou muito pequeno para ser armazenado no espaço reservado para ele, ocorrem os fenômenos chamados de overflow e underflow respectivamente. Outro fenômeno é o próprio overflow da mantissa como mostrado acima.

3.7.4 Adição e Subtração em Ponto Flutuante

A adição e a subtração de ponto flutuante tratam os expoentes junto com o valor dos operandos, logo há a necessidade de equalizar os expoentes e efetuar a operação sobre a mantissa equalizada e o resultado deve ser normalizado para a representação utilizada:

Vamos analisar a seguinte adição em Ponto Flutuante:

01101010 + 01011100 = ?

Processo:

Mantissa 1 = 1010	Expoente = 110 (2 na Notação de Excesso)
Mantissa 2 = 1100	Expoente = 101 (1 na Notação de Excesso)

Equalizando os expoentes, temos:

$$\begin{array}{r} 1.0100 \\ +\ \underline{0.1100} \\ 10.0000 \end{array}$$

Normalizando:

Resultado = 1000
Expoente = 110 (2 na Notação de Excesso)

Representação do resultado (01101010 + 01011100) em Ponto Flutuante = 01101000

3.8 Lógica Binária

George Boole publicou a álgebra booleana (em 1854), sendo um sistema completo que permitia a construção de modelos matemáticos para o processamento computacional. O fascinante na lógica booleana é partir de três operadores básicos, que veremos a seguir, e construir **Circuitos Lógicos** capazes de realizar as diversas operações necessárias para um computador.

A seguir a descrição de cada um dos 3 operadores básicos. A Figura 3.8 [45] representa os valores da tabela de valores (Tabela Verdade) dos operadores e a representação gráfica, sendo também chamadas de portas lógicas.

3.8.1 Operador NOT

O operador unário NOT, negação binária, resulta no complemento do operando, ou seja, será um bit `1` se o operando for `0`, e será `0` caso contrário, conforme podemos confirmar pela tabela de verdade, onde A é o bit de entrada e S é a resposta, ou bit de saída.

3.8.2 Operador AND

O operador binário AND, ou conjunção binária devolve um bit 1 sempre que ambos operandos sejam `1`, conforme podemos confirmar pela tabela de verdade, onde A e B são bits de entrada e S é o bit-resposta, ou bit de saída.

3.8.3 Operador OR

O operador binário OR, ou disjunção binária devolve um bit `1` sempre que pelo menos um dos operandos seja `1`, conforme podemos confirmar pela tabela de verdade, onde A e B são os bits de entrada e S é o bit-resposta, ou bit de saída.

AND

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

NOT

ou

A	S
0	1
1	0

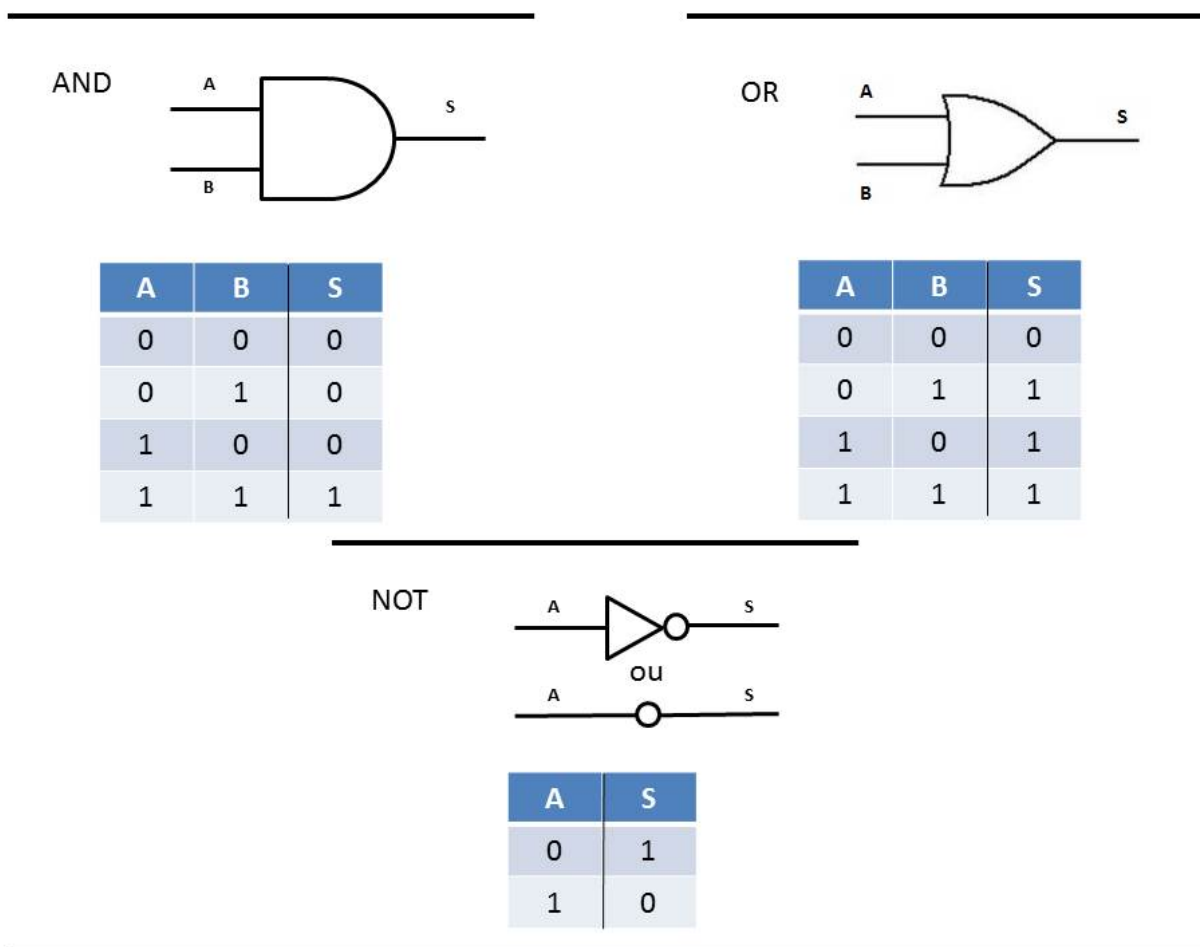



Figura 3.8: Representação gráfica dos operadores lógicos AND, OR e NOT, com seus valores de entrada e saída.

3.8.4 A soma em um Computador

Neste capítulo, aprendemos sobre os sistemas de numeração, dando ênfase ao sistema binário, sendo este o sistema adotado pelos computadores. Aprendemos como funciona a aritmética binária, soma, subtração, representação negativa dos números, entre outros. Em seguida, foi apresentada a lógica binária e seus três comandos básicos (AND, OR, NOT). Mas como um computador soma de fato?

Primeiramente, precisamos abordar as portas lógicas, elas são a base para as outras operações. A construção de uma porta lógica, utiliza conhecimentos de circuitos eletrônicos formados por diodos, resistências, capacitores entre outros que são abordados em cursos avançados da Eletrônica Analógica, entretanto, seu entendimento foge ao escopo deste livro.

Dica

Para melhores detalhes sobre portas lógicas visite: http://www.dcmm.puc-rio.br/cursos/eletronica/html/sld001.htm.

O importante agora é sabermos que existem portas lógicas que seguem a lógica binária já apresentada e que estas portas podem ser combinadas, formando os **Circuitos Digitais**. A Figura Figura 3.9 [46] apresenta um Circuito Digital Somador de Dois Bits.

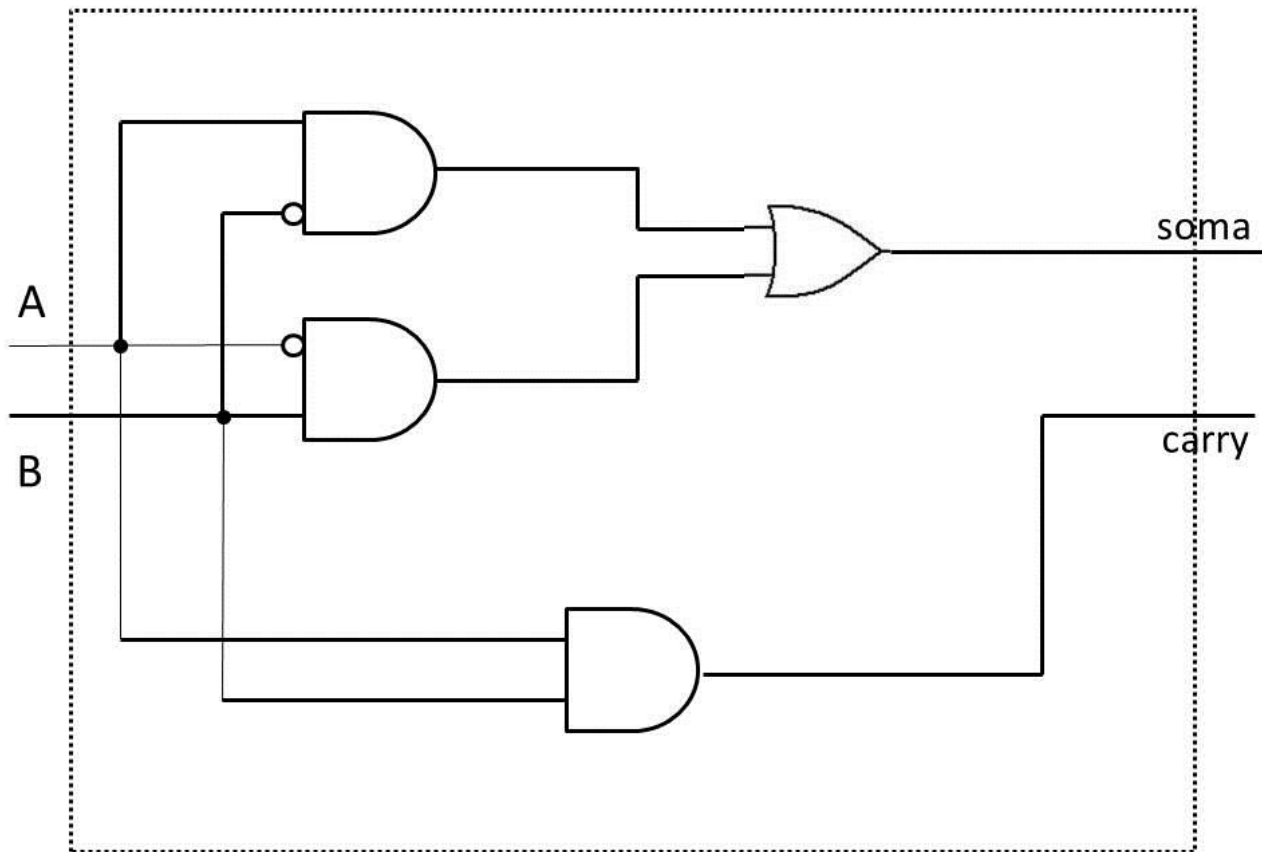



Figura 3.9: Circuito Digital Somador de Dois Bits formado pelas portas lógicas básicas (AND, OR, NOT).

Propomos ao leitor, tentar acompanhar passo a passo o circuito digital proposto e entender como possíveis entradas binárias em A e B terão o resultado esperado nas saídas *soma* e *carry*. Em seus testes utilize a Tabela 3.4 [46], e se tiver alguma dúvida sobre os valores da tabela, revisem a operação de soma com dois bits, onde a saída *soma* representa o valor da soma na unidade corrente e o *carry* representa o "vai um" da operação.

Tabela 3.4: Tabela de valores da operação de Soma de Dois Bits

A	B	soma	carry (vai um)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figura 3.10: Vídeo sobre o Circuito Digital Somador de 2 Bits: http://youtu.be/E5yDNF2clQw

3.9 Recapitulando

Neste capítulo estudamos a origem dos sistemas de numeração. Aprendemos a trabalhar com os sistemas de numeração mais utilizados na computação. Aprendemos mais profundamente o funcionamento do sistema binário, sua aritmética, representação de números negativos, representação de números fracionários e a lógica binária proposta por George Boole.

Por fim demos uma introdução sobre como o computador consegue realizar as operações aritméticas a partir de porta lógica básica, demonstrando o poder da matemática no auxílio dos procedimentos computacionais.

3.10 Atividades

1. Efetue as seguintes conversões:
   a. Converta para decimal 110101_2 e 1001_2
   b. Converta para octal 110111011101_2 e 1111111_2
   c. Converta para hexadecimal 101100101100_2
   d. Converta para binário $FF1F_{16}$ e ABC_{16}
2. Efetue as seguintes conversões:
   a. Converta para decimal 1101.01_2 e 10.01_2 (Ponto Fixo)
   b. Converta para octal 110111011101_2 e 1111111_2
   c. Converta para hexadecimal 101100101100_2
   d. Converta para binário 0xFF1F (Hexadecimal[1])
3. Converta o número –5 para uma representação binária usando 4-bits, com as seguintes representações:

[1]Em programação costumamos representar números hexadecimais iniciando-o com "0x", portanto 0xFF1F equivale a $FF1F_{16}$.

a. Sinal e amplitude
b. Complemento para 1
c. Complemento para 2
d. Notação de Excesso

4. Converta o número –33 para uma representação binária usando 6-bits, com as seguintes representações:

   a. Sinal e amplitude
   b. Complemento para 1
   c. Complemento para 2

5. Converta para decimal o valor em binário (usando apenas 5-bits) 10101_2, considerando as seguintes representações:

   a. Inteiro sem sinal
   b. Sinal e amplitude
   c. Complemento de 2

6. Efetue os seguintes cálculos usando aritmética binária de 8-bits em complemento de 2, ou seja, primeiro converta o valor para binário e depois efetue a operação aritmética.

   a. $4_{10} + 120_{10}$
   b. $70_{10} + 80_{10}$
   c. $100_{10} + (-60_{10})$
   d. $-100_{10} - 27_{10}$

7. A maioria das pessoas apenas consegue contar até 10 com os seus dedos; contudo, os engenheiros informáticos podem fazer melhor! Como? Cada dedo conta como um bit, valendo 1 se esticado, e 0 se dobrado.

   a. Com este método, até quanto é possível contar usando ambas as mãos?
   b. Considere que um dos dedos na extremidade da mão é o bit do sinal numa representação em complemento para 2. Qual a gama de valores que é possível representar com ambas as mãos?

8. Efetue as operações Aritméticas no sistema binário:

   a. $100101010_2 + 101010111_2$
   b. $101010110_2 - 010110111_2$
   c. $100000000_2 - 000010011_2$
   d. $111111111_2 + 101010101_2$

9. Converta para a representação em Ponto Flutuante, com 12 bits (1: sinal; 4: expoente; 8: mantissa), os seguintes valores, dados em base 10 (apresente todos os cálculos):

   a. +12
   b. –10.75
   c. – 8.25

Capítulo 4

Organização e Funcionamento do Computador

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Descrever a arquitetura geral de um computador
- Descrever seus componentes básicos e suas funções, como por exemplo a UCP, a memória principal, e os dispositivos de E/S
- Descrever o ciclo de máquina responsável pela execução de programas de computador

Um sistema de computador é integrado pelo seu hardware e seu software. O hardware é o equipamento propriamente dito, e seus elementos básicos são: unidade central de processamento, memória principal e seus dispositivos de entrada e saída.

O software é constituído pelos programas que lhe permitem atender às necessidades dos usuários. Ele abriga programas fornecidos pelos fabricantes do computador e programas desenvolvidos pelo usuário.

Neste capítulo, iremos identificar como estes dois elementos unidos permitem que atividades pré-programadas, através de uma linguagem, sejam executas de forma automática pelo computador.

4.1 Arquitetura de um Computador

Os circuitos de um computador que executam operações sobre dados, tais como adição e subtração, são isolados em uma região chamada Unidade Central de Processamento UCP (CPU – *Central Processing Unit*), ou processador.

Os dados que estão armazenados na memória principal do computador são transferidos através de barramentos que interligam estes componentes.

A comunicação com o mundo externo, os usuários, se dá pelos dispositivos de Entrada e Saída (E/S). A comunicação entre o computador e estes dispositivos se dá através dos controladores de cada dispositivo de E/S. Em computadores comuns, estes controladores correspondem placas de circuito encaixadas na placa principal do computador (placa mãe). Está ilustrada na Figura 4.1 [50], a arquitetura básica de um computador, demonstrando a organização de seus componentes básicos.

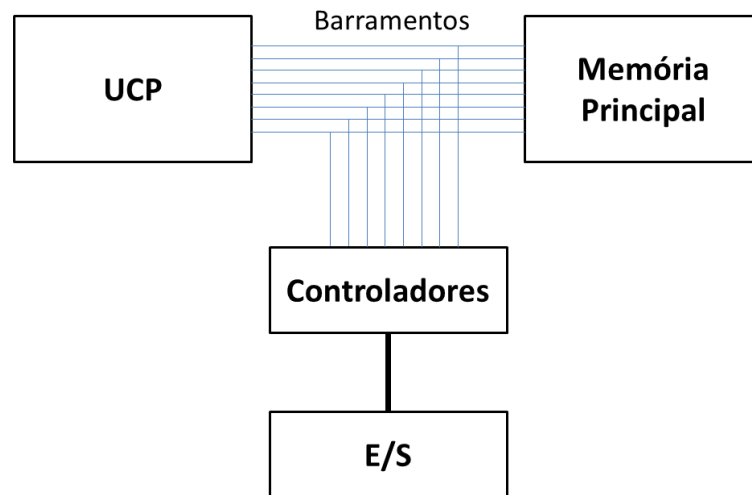



Figura 4.1: Arquitetural geral de um computador

Esta seção apresenta uma descrição sobre cada unidade desta Arquitetura, descrevendo seus componentes e funções básicas.

4.1.1 Memória Principal

A memória do computador consiste numa coleção de registradores numerados consecutivamente (endereçados), onde cada um possui um tamanho denominado de **tamanho da palavra**, que pode variar em 16, 32, 64 e 128 bits, com a palavra de 32 bits sendo a mais comum hoje em dia, e a palavra de 64 bits aumentando de popularidade.

Cada registrador tem um endereço, chamado de localização na memória, estas são organizadas linearmente em ordem consecutiva. O número único que identifica cada palavra é chamado de endereço.

A memória possui um espaço de endereçamento representado pelo tamanho em bits do seu endereço, logo, um espaço de endereçamento de 32 bits pode acessar qualquer palavra de memória em qualquer lugar no intervalo de 0 a 2^{32}-1.

O espaço de endereçamento pode ser dividido em regiões distintas usadas pelo sistema operacional, dispositivos de E/S, programas de usuário e pilha do sistema operacional.

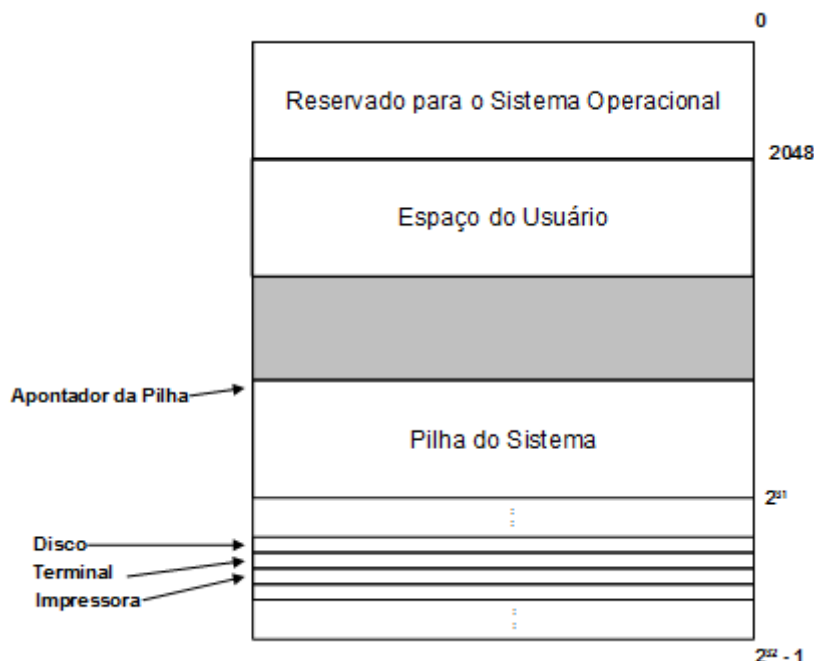



Figura 4.2: Mapa de Memória do Computador

As regiões ilustradas na Figura 4.2 [51] compõem um possível mapa de memória. Os endereços acima de 2048 são reservados para uso do sistema operacional. O espaço do usuário é onde um programa do usuário será carregado. A porção do espaço de endereçamento entre 2^{31} e $2^{32} - 1$ está reservada para dispositivos de E/S.

É importante manter clara a distinção entre o que é endereço e o que é dado. Uma palavra na memória, pode ter distintas representações dependendo do seu uso. Ela pode armazenar uma instrução contendo a operação e os operandos (dados de entrada) para a realização de uma específica operação, mas também pode armazenar o endereço de uma outra região de memória. Logo, o endereço é um apontador para uma posição de memória que contém dados, e estes são informações significativas para a realização de alguma atividade no computador, ou a representação de alguma informação.

4.1.2 Unidade Central de Processamento (UCP)

A Unidade Central de Processamento, ilustrada na Figura 4.3 [52], é composta por duas partes principais: a **unidade lógica e aritmética** (ULA), formada por circuitos que manipulam os dados através de operações binárias (dois operandos) e unárias (um operando). Exemplos incluem a *soma* e operadores lógicos: *and*, *or* e *not*. E a **unidade de controle**, cujos circuitos são responsáveis por coordenar as operações da UCP.

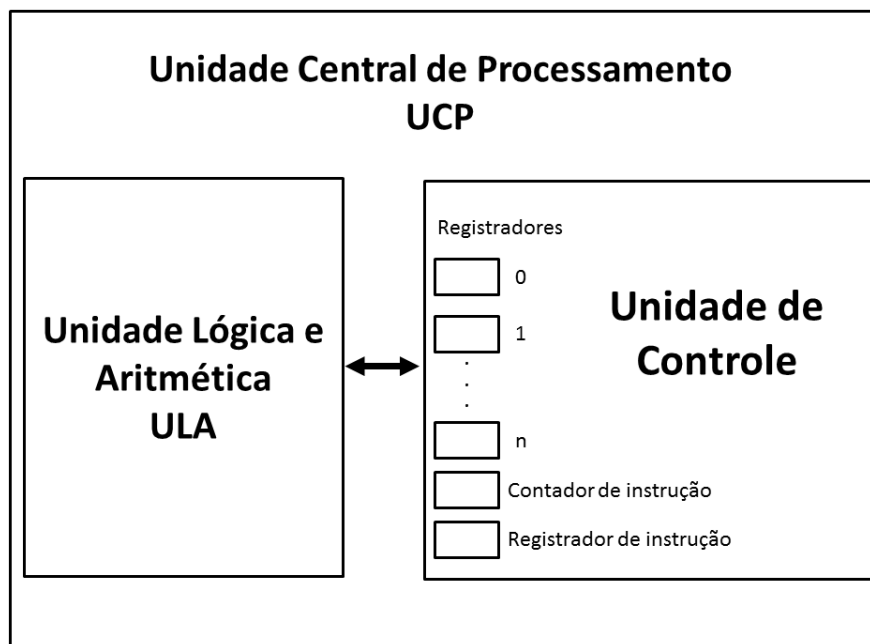



Figura 4.3: Componente lógicos da UCP

Para o armazenamento e a comunicação entre estas duas unidades a UCP contém circuitos de armazenamento chamados de registradores, que se assemelham às células de armazenamento da memória principal.

Alguns registradores funcionam como posições intermediárias de armazenamento para os dados manipulados pela UCP. Nestes registradores são armazenados os dados de entrada para a ULA e ainda proporcionam um local de armazenamento para o resultado das operações.

Os dados a serem manipulados pela ULA tem origem na memória principal, sendo de responsabilidade da unidade de controle transferir estes dados aos registradores, informar à ULA sobre quais registradores estão os dados de entrada, ativar o circuito da operação apropriada e informar em que registrador deve guardar o resultado da operação.

A transferência desta informação oriunda da memória principal se dá através do **barramento** que é responsável por transmitir padrões de bits entre a UCP, os dispositivos de E/S e a memória principal.

Nota

Executar uma simples operação de soma é mais complexo que apenas somar estes números. Coordenado pela unidade de controle, os registradores intermediam a comunicação da memória principal e a ULA. Este processo pode ser resumido assim:

PASSOS

1. Obter da memória um dos valores da soma e guardar em um registrador;
2. Obter da memória o outro número a ser somado e armazená-lo em outro registrador;
3. Acionar o circuito de adição tendo os registradores do passo 1 e 2 como entrada, e escolher outro registrador para armazenar o resultado;

4. Armazenar o resultado na memória principal;
5. Finalizar operação.

4.1.3 Unidades de Entrada/Saída

Entrada/Saída (E/S) compreende todas as maneiras como o computador se comunica com os usuários e outras máquinas ou dispositivos. Os dispositivos de entrada aceitam dados e instruções do usuário, os dispositivos de saída retornam os dados processados.

Os dispositivos de saída mais comuns são a tela de vídeo, conhecida como monitor, e a impressora. Os dispositivos de entrada mais conhecidos são teclado e mouse. Os sistemas de multimídia possuem alto-falante como saída e microfone como entrada adicional.

Os dispositivos de E/S trabalham com a memória do computador do seguinte modo: os dados captados pelos dispositivos de entrada são representados em pulsos elétricos e transmitidos ao computador, ali estes pulsos são convertidos em dados binários e armazenados na memória do computador. No caminho inverso, a informação binária é transformada em pulso elétrico e encaminhada para o dispositivo de saída especialista para tratá-lo e gerar uma saída ao usuário.

Um dispositivo especial de E/S de um computador é o disco rígido (HD), nele são armazenados todos os dados que devem persistir num sistema computacional, mesmo na ausência de energia. Todos os programas que não estão em execução se encontram no disco, seu único problema é o tempo excessivo para a recuperação e escrita de uma informação, havendo assim a necessidade de se trabalhar com a memória volátil (memória principal), mais rápida, porém mais cara.

4.1.4 O Modelo de Barramento

O objetivo do barramento é reduzir o número de interconexões entre a UCP e seus subsistemas. Em lugar de mantermos um caminho de comunicação entre a memória e cada um dos dispositivos de entrada e saída, a UCP é interconectada com os mesmos via barramento de sistema compartilhado.

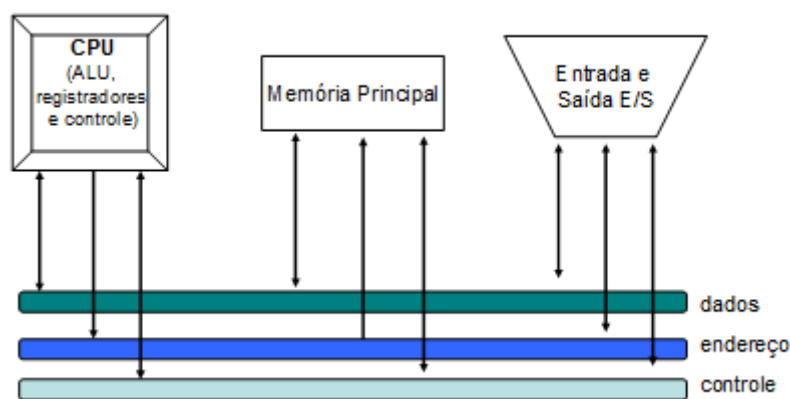



Figura 4.4: Modelo de Barramento do Computador

Os componentes são interconectados ao barramento da forma ilustrada na Figura 4.4 [53]. A UCP gera endereços que são colocados no **barramento de endereços**, e a memória recebe endereços do mesmo. O caminho inverso desta operação não é possível, como pode ser observado na figura.

Durante a execução de um programa, cada instrução é levada até à ULA (Unidade Lógica e Aritmética) a partir da memória, uma instrução de cada vez, junto com qualquer dado que seja necessário para executá-la. A saída do programa é colocada em um dispositivo, tal como display de vídeo ou disco. A comunicação entre os três componentes (UCP, memória e E/S) é feita sempre pelos barramentos.

4.2 Programando um computador

A grande evolução na arquitetura dos computadores foi a flexibilidade da unidade de controle quanto ao tratamento de instruções. Em suas primeiras versões existiam circuitos físicos montados na unidade de controle que realizavam algumas atividades específicas, e sempre que o usuário quisesse utilizar outras operações, era necessário reprogramar fisicamente a fiação do circuito para que a nova operação pudesse ser executada.

O leitor deve perceber que esta estratégia tornaria qualquer programação inviável para leigos da área de circuitos eletrônicos. Esta seção apresenta o funcionamento atual da UCP para executar os programas do sistema e de usuários.

4.2.1 Linguagem de Máquina

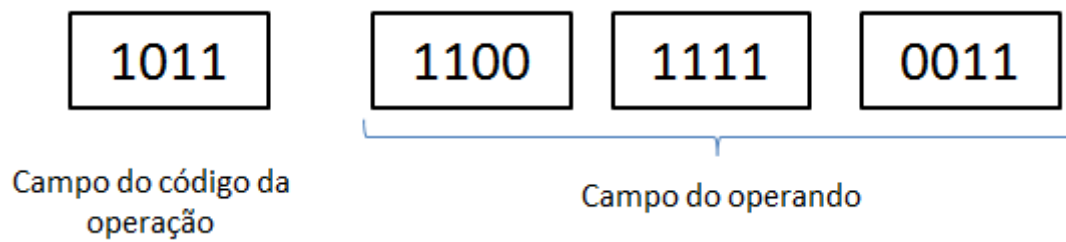
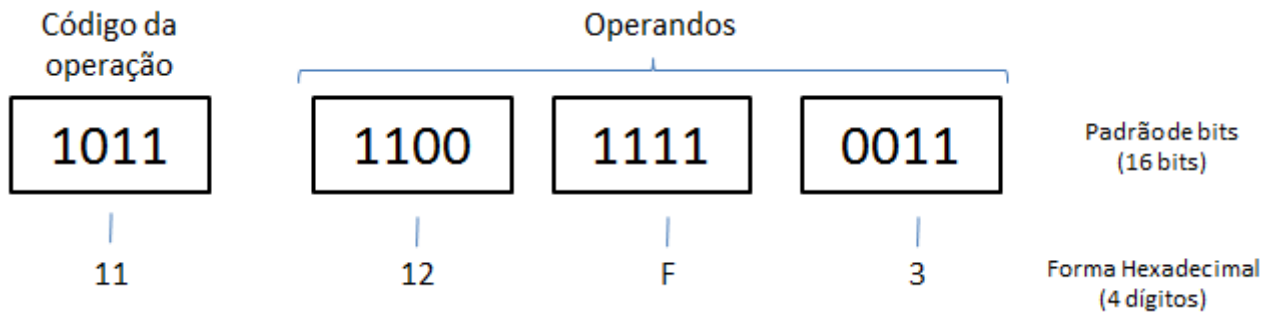
A unidade de controle moderna é projetada para reconhecer instruções codificadas como padrões de bits. Ao conjunto de instruções é dado o nome de linguagem de máquina, e cada uma destas instruções é chamada de instrução de máquina.

Algo que pode surpreender o leitor é a pequena quantidade necessária para a UCP decodificar, sendo através da combinação destas poucas instruções, bem planejadas, que o computador desenvolve tarefas de usos gerais nas mais diversas situações em nosso cotidiano. E a grandeza da Ciência da Computação é que se tivermos projetado instruções bem planejadas, a adição de outras instruções mais específicas se torna desnecessária, pois uma combinação das instruções básicas tem o mesmo efeito lógico da instrução específica.

A decisão entre instruções específicas ou instruções de uso geral levou a disputa de duas filosofias de projetos para as UCPs. A primeira, chamada computador com **conjunto mínimo de instruções** (*Reduced Instruction Set Computer* - RISC), tem em sua defesa o fato de criar máquinas simples e eficientes. O contraponto é a filosofia do computador com **conjunto de instruções complexas** (*Complex Instruction Set Computer* - CISC), que tem como argumento a facilidade de programação, já que uma única instrução é capaz de realizar várias tarefas individuais do RISC.

As duas filosofias tem entrada no mercado, sendo o CISC adotado pela família Pentium da Intel e o RISC adotado pela Apple Computer, IBM e Motorola.

Existem três tipos de instruções: as instruções de transferência de dados, que realizam cópia de valores entre registradores da UCP e a memória principal, como por exemplo `STORE` e `LOAD`; as instruções lógicas e aritméticas que ativam os circuitos específicos da ULA para a realização das operações, como por exemplo `ADD`, `SHIFT`, `OR`, `AND` e etc.; e por fim, as instruções de controle, responsáveis por tratar a sequência da execução do programa sem haver manipulação de seus dados, como por exemplo o `JUMP` e `CALL`, usadas para a mudança do fluxo normal de um programa, implementando assim os desvios condicionais, códigos de repetição, chamada de função e retorno.

A codificação de uma instrução é composta de duas partes, o **campo código da operação** e o **campo do operando**.

Podemos observar no exemplo a seguir, a codificação da operação `STORE` e seus operandos:

O primeiro dígito hexadecimal, o `11`, representa, neste exemplo, a operação `STORE` (armazena o conteúdo de um registrador em memória). O dígito hexadecimal seguinte representa o identificador do registrador (valor `12`) que possui o conteúdo a ser gravado, e já o par de dígitos hexadecimais `F3` representa o endereço na memória principal onde o conteúdo do registrador `12` será guardado. Podemos traduzir este código da seguinte forma:

Armazene o padrão de bits contido no registrador `12` para a célula de memória de endereço `F3`.

4.2.2 Executando Programas em Linguagem de Máquina

Um programa é uma sequência de instruções em uma linguagem a ser executada com o objetivo de realizar uma determinada atividade pré-programada.

O programa em linguagem de máquina fica posto na memória principal, sendo de responsabilidade da unidade de controle, a busca por cada instrução de máquina, sua decodificação e o gerenciamento da execução desta instrução com o auxílio da ULA. Após o término de cada instrução o processo se repete dando continuidade ao ciclo de máquina, este ciclo está ilustrado na Figura 4.5 [56].

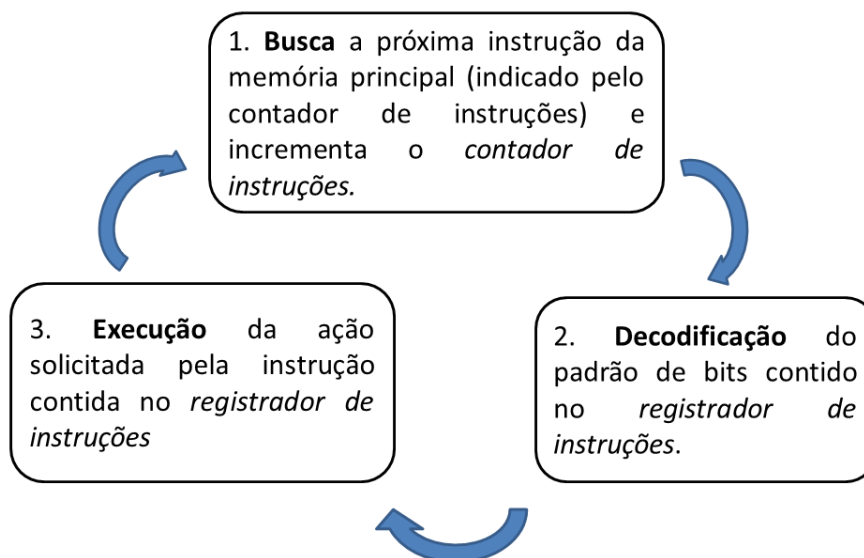



Figura 4.5: As três fases do ciclo de máquina

Para o controle deste ciclo a unidade de controle possui dois registradores de propósito específico: o **contador de instruções** e o **registrador de instruções**. No contador de instruções é armazenado o endereço de memória da próxima instrução a ser executada, assim a unidade de controle fica informada sobre a posição do programa em execução. Já o registrador de instruções guarda a instrução de máquina que está em execução.

Como ilustrado na Figura 4.5 [56], o ciclo de máquina é divido em três fases. Em cada fase a unidade de controle utiliza seus registradores para auxiliá-la na execução das instruções:

Busca
: a unidade de controle pede para a memória principal transferir a instrução contida no endereço de memória indicado pelo contador de instruções e este conteúdo é armazenado no registrador de instruções. Por fim, o conteúdo do contador de instruções é incrementado, ficando pronto para a próxima fase de busca;

Decodificação
: a unidade de controle separa os campos da instrução contida no registrador de instruções de acordo com o tipo de operação, identificando os operandos da instrução;

Execução
: os circuitos específicos da ULA são ativados, os dados de entrada carregados e a tarefa indicada na instrução é executada.

Ao final da execução, a unidade de controle recomeçará o ciclo, partindo da fase de busca.

4.3 Recapitulando

Neste capítulo estudamos a Arquitetura básica de um computador, identificamos cada um dos seus componentes, descrevendo suas funcionalidades principais e um resumo de seu funcionamento interno. Descrevemos como estes componentes interagem entre si para o funcionamento do computador, dentre estes componentes estudamos a Unidade Central de Processamento, a Memória Principal, os dispositivos de Entrada e Saída e os Barramentos.

Por fim, demos uma introdução sobre como o computador consegue executar um programa desenvolvido em linguagem de máquina, apresentando, para isso, o conceito de ciclo de máquina.

4.4 Atividades

1. Qual o papel de cada unidade dentro da UCP?
2. Qual a função dos registradores existentes na UCP?
3. Qual a importância dos Barramentos na Arquitetura de um computador?
4. Como a unidade de controle gerencia o ciclo de máquina de um computador?
5. Descreva o caminho percorrido entre os componentes de um computador para que uma soma de dois operandos tenha seu resultado apresentando em um monitor de vídeo.

Capítulo 5

Algoritmos, Linguagem de Programação, Tradutor e Interpretador

OBJETIVOS DO CAPÍTULO

Ao final deste capítulo você deverá ser capaz de:

- Apresentar o conceito de algoritmos;
- Apresentar como os algoritmos são formalizados nos computadores através de linguagens de programação;
- Descrever os softwares básicos que permitem ao computador entender os algoritmos descritos nas linguagens de programação.

5.1 Algoritmos

Historiadores trazem divergências sobre a origem da palavra algoritmo, sendo a mais difundida, devido ao seu sobrenome, a de Mohamed ben Musa Al-Khwarizmi, um matemático persa do século IX, cujas obras foram traduzidas no ocidente no século XII, tendo uma delas recebido o nome *Algorithmi de numero indorum* (indiano), acerca dos algoritmos que trabalham sobre o sistema de numeração decimal.

Independente de sua real etimologia, a ideia principal contida na palavra refere-se à descrição sistemática da maneira de se realizar alguma tarefa. Para a Ciência da computação, o conceito de algoritmo foi formalizado em 1936 por Alan Turing (Máquina de Turing) e Alonzo Church, que formaram as primeiras fundações da Ciência da computação. Sendo esta formalização descrita a seguir:

> Um algoritmo é um conjunto não ambíguo e ordenado de passos executáveis que definem um processo finito.

Em sua definição o algoritmo requer um conjunto de passos ordenados, isto significa que estes passos devem ser bem definidos e estruturados para uma ordem de execução. Isto não quer dizer que estes passos devem ser executados sempre em uma única sequência consistindo de um primeiro passo seguido por um segundo, e assim por diante. Muitos algoritmos, conhecidos como algoritmos paralelos, contém mais do que uma sequência de passos, cada um sendo projetado para executar um processo distinto em máquinas multiprocessadas. Logo, as sequências dos passos podem intercalar

entre si dependendo do ambiente de execução, entretanto, dentro de uma mesma sequência sua ordem de execução não muda durante o processo.

Seguindo a definição, todo algoritmo deve consistir em passos executáveis. Considere a seguinte instrução

> Faça uma lista de todos os números inteiros ímpares

Sendo um algoritmo para sua solução impossível de ser realizado, pois existe uma infinidade de números inteiros ímpares. Logo, um algoritmo deve ser eficaz, ou seja, que possa ser resolvido. Podemos criar um algoritmo para solucionar a instrução acima modificando sua descrição

> Faça uma lista de todos os números inteiros ímpares no intervalo [1; 100]

Por fim, os passos em um algoritmo não podem ser ambíguos, isto significa que durante a execução de um algoritmo, o estado atual do processamento deve ser suficiente para determinar única e completamente as ações requeridas por cada passo.

5.1.1 Exemplo de um Algoritmo

Guilherme recebe alguns convidados que estão visitando a cidade em sua casa e precisa ensiná-los a chegar à igreja para a missa do domingo. O anfitrião muito organizado apresenta o mapa de seu bairro como visto na Figura 5.1 [60] e propõe o seguinte algoritmo para que seus amigos não se percam nas ruas da cidade.

Algoritmo

```
Tire o carro da garagem
Pegue a rua à esquerda
Siga em frente
Pegue a primeira rua à direita
Siga em frente
Pegue a primeira rua à esquerda
Siga em frente
Pegue a primeira à direita
Siga em frente
Procure a igreja no lado esquerdo
Estacione em frente à igreja
```

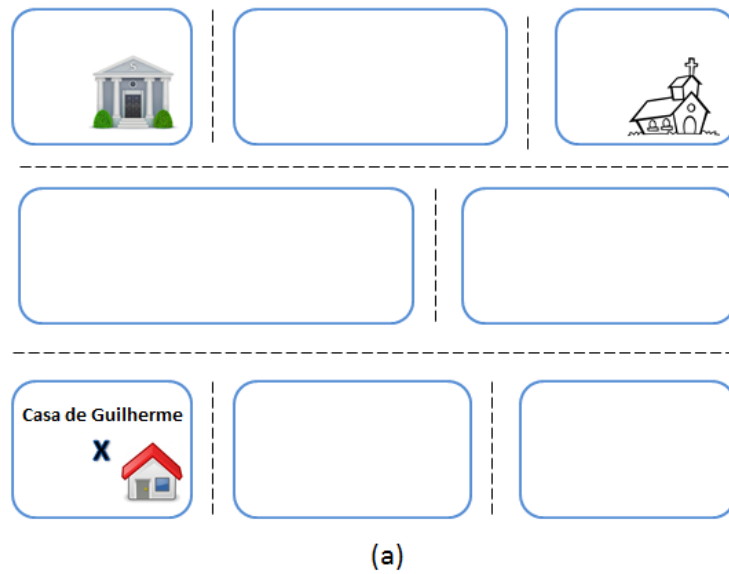



Figura 5.1: Mapa da cidade de Guilherme

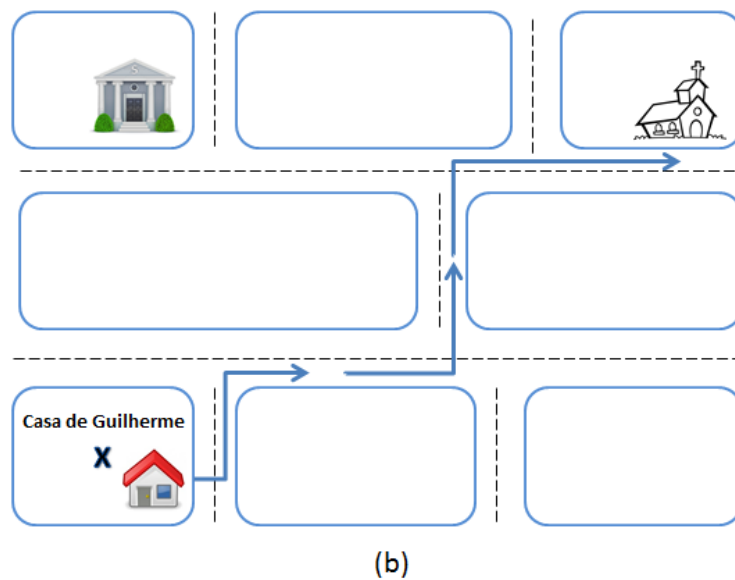



Figura 5.2: Rota para igreja descrita no algoritmo

Neste exemplo, os passos são descritos em sua ordem de execução de forma concisa e sem dubiedades em seu entendimento.

Nota

Que tal fazer um algoritmo? Como seria o algoritmo para Guilherme ensinar seus amigos a chegarem no banco?

5.1.2 Programa de Computador

Um programa de computador é essencialmente um algoritmo que diz ao computador os passos específicos e em que ordem eles devem ser executados, como por exemplo, os passos a serem tomados para calcular as notas que serão impressas nos boletins dos alunos de uma escola.

Quando os procedimentos de um algoritmo envolvem o processamento de dados, a informação é lida de uma fonte de entrada, processada e retornada sob novo valor após processamento, o que geralmente é realizado com o auxílio de um conjunto de instruções e estrutura de dados.

Figura 5.3: (a) Autômato suíço do século XIX escrevendo um poema

Figura 5.4: (b) Autômato do filme “A Invenção de Hugo Cabret”

Algoritmos podem ser implementados em circuitos elétricos ou até mesmo em dispositivos mecânicos (autômatos, vide Figura 5.3 [61] e Figura 5.4 [61]). Mania dos inventores do século XIX, os autômatos eram máquinas totalmente mecânicas, construídas com a capacidade de serem programadas para realizar um conjunto de atividades autônomas. Em 2011, o filme *A Invenção de Hugo Cabret* (tradução brasileira) do cineasta Martin Scorsese traz a história do ilusionista Georges Méliès precursor do cinema e um colecionador de autômatos, sendo uma de suas máquinas o fio condutor desta história. O autômato específico era capaz de desenhar a cena emblemática do seu filme "Viagem à Lua".

Entretanto, a maioria dos algoritmos são desenvolvidos para programas de computador, para isto, existe uma grande variedade de linguagens de programação, cada uma com características específicas que podem facilitar a implementação de determinados algoritmos ou atender a propósitos mais gerais, definiremos melhor uma linguagem de programação na Seção 5.2 [63].

5.1.3 Construindo um algoritmo em sala de aula

Imagine que o leitor queira ensinar o conceito de algoritmo para um grupo de alunos. Uma forma muito interessante de abordar a construção de um primeiro algoritmo em sala é apresentar um jogo que utilize uma sequência de passos lógicos para sua resolução. Peça para os alunos pensarem em uma solução e definir comandos em português a serem seguidos pelo seu colega ao lado para resolver o problema através de sua solução proposta. Se o colega obtiver êxito na resolução do jogo, seu algoritmo terá sido validado.

Propomos a abordagem do conhecido jogo “imprensadinho”, onde o jogador tem o objetivo de adivinhar um número escolhido aleatoriamente pelo seu adversário dentro de um intervalo de valores pré-estabelecido. O jogador pergunta por qualquer número dentro deste intervalo e o adversário tem que responder se o número escolhido foi descoberto ou não. O jogador ainda pode perguntar se o número a ser encontrado é maior ou menor que o número corrente testado. O jogo acaba quando o jogador descobre o número escolhido.

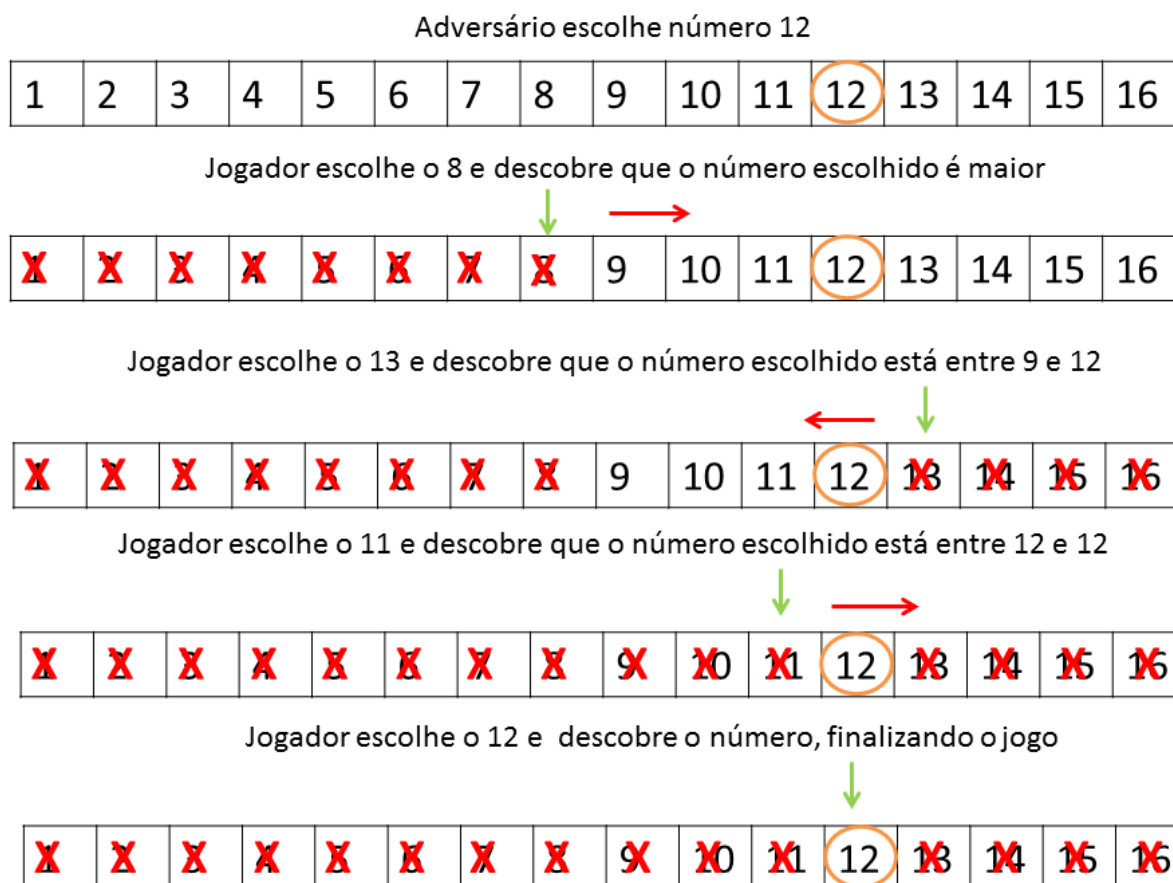



Figura 5.5: Proposta de algoritmo para o “imprenssadinho”

> **Nota**
> Antes de propor esta atividade a seus alunos, que tal tentar elaborar um algoritmo que formalize sua solução. Descreva testes a serem efetuados e instruções para descrever as decisões a serem tomadas.

5.2 Linguagem de Programação

Imagine aplicações para redes sociais como o Facebook, jogos eletrônicos e decodificadores de vídeo digital sendo implementados em linguagem de máquina (linguagem binária), seria uma tarefa impossível. Para isso, linguagens de programação foram desenvolvidas para permitir que algoritmos sejam aceitáveis para os humanos e facilmente convertidos em instruções de linguagem de máquina. Podemos definir uma linguagem de programação como um conjunto de palavras (vocabulário) e de regras (sintaxe) que permite a formulação de instruções a um computador.

5.2.1 Primeiras gerações

Os códigos interpretados pelos primeiros computadores consistiam em instruções codificadas como dígitos numéricos. Escrever programas nesta linguagem, além de ser tedioso, é passível de muitos erros que devem ser localizados e corrigidos para finalizar o trabalho.

Na década de 40, pesquisadores desenvolveram um sistema de notação onde instruções numéricas podem ser representadas por mnemônicos. Por exemplo, a instrução

```
Mova o conteúdo do registrador 3 para o registrador 1
```

expressa numericamente como:

```
4056
```

usando uma linguagem de máquina. Já em um sistema mnemônico podemos representar esta instrução como:

```
MOV   R5,   R6
```

Com o uso do sistema mnemônico, programas chamados montadores (*assemblers* em inglês) foram desenvolvidos para converter expressões mnemônicas em linguagem de máquina. Por isso, muitas vezes as linguagens mnemônicas são conhecidas como linguagem assembly.

Apesar da melhoria acarretada com a adoção do sistema mnemônico, sua programação ainda traz muitos dissabores aos desenvolvedores. A linguagem é uma troca direta de comandos básicos da linguagem de máquina, tornando a sua programação totalmente amarrada a arquitetura da máquina em que o código está sendo desenvolvido (dependência de plataforma). E a filosofia de desenvolvimento era toda baseada em comandos de mais baixo nível da máquina. Fazendo uma analogia com a construção de uma casa, seria necessário pensar em sua construção a partir de tijolos, canos, cimento, pedra e etc. Embora toda construção precise trabalhar com estes elementos básicos, durante o projeto de uma casa, o arquiteto pensa em termos de salas, varanda, portas e etc.

Com esta filosofia os pesquisadores de computação desenvolveram a terceira geração de linguagens de programação, sendo suas primitivas básicas de alto nível e independentes de máquina, um diferencial das linguagens anteriores. Estas linguagens são desenvolvidas para usos mais especialisados. Os exemplos mais conhecidos desta primeira fase das linguagens de alto nível são o FORTRAN (FORmula TRANslation), que foi desenvolvido para aplicações científicas e de engenharia, e o COBOL (Common Business-Oriented Language) desenvolvido para aplicações comerciais.

O objetivo das linguagens de alto nível era descrever operações sem se preocupar quais instruções de máquina seriam necessárias para implementar estas operações. Uma vez identificado este conjunto de primitivas de alto nível necessárias para o algoritmo, um programa, conhecido como tradutor, é acionado para converter programas escritos na linguagem de alto nível, em programas com linguagem de máquina. Este software entre outros serão melhor detalhados na Seção 5.3 [64].

5.2.2 Paradigma de Programação

Um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa. Os paradigmas representam abordagens fundamentalmente diferentes para a construção de soluções para os problemas, portanto afetam todo o processo de desenvolvimento de software. A seguir serão descritos os paradigmas de programação clássicos:

Paradigma imperativo
Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa. Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens como visto no algoritmo das rotas.

Paradigma declarativo
Descreve propriedades da solução desejada, não especificando como o algoritmo em si deve agir. Muito popular em linguagens de marcação, sendo utilizado na programação das páginas web (linguagem HTML) e descrição de documentos multimídia como a linguagem Nested Context Language – NCL, adota pelo padrão brasileiro de TV Digital.

Paradigma funcional
Trata a computação como uma avaliação de funções matemáticas. Ela enfatiza a aplicação de funções, em contraste da programação imperativa, que enfatiza mudanças no estado do programa. Neste paradigma ao se pensar em uma função simples de calculo de médias de notas, usamos o auxílio de funções mais primitivas, podendo a função Media (Numeros) ser representada pela expressão:

```
(Divide  (Soma Numeros)  (Conta Numeros) )
```

logo a função Divide opera com os resultados das funções Soma e Conta.

Paradigma orientado a objeto
Neste paradigma, diferente do paradigma imperativo, os dados passam a ter um papel principal na concepção do algoritmo. No paradigma imperativo, rotinas de controle manipulam os dados que são elementos passivos. Já na orientação a objetos, os dados são considerados objetos auto gerenciáveis formados pelos próprios dados e pelas rotinas responsáveis pela manipulação destes dados.

5.3 Tradutor e Interpretador

Ao receber uma bicicleta no natal Carlinhos precisa ler o manual de instruções e seguir passo a passo as tarefas descritas no documento para poder se divertir com seu presente. Podemos dizer que Carlinhos é um interpretador dos comandos fornecidos pelo manual de instruções. Entretanto seu pai encontrou uma promoção na internet e comprou um produto fabricado na França e o menino ao se deparar com o manual percebeu que o mesmo não poderia ser "interpretado" já que não sabia ler em francês. Para resolver o problema seu pai contratou um tradutor de francês para português, assim, este novo manual pôde ser "interpretado" por Carlinhos e enfim sua bicicleta seria montada.

No computador, o problema de Carlinhos se repete diariamente, havendo a necessidade de softwares básicos para traduzir e interpretar os diversos programas dos usuários escritos em diversas linguagens existentes. Os softwares que convertem um programa de usuário escrito em uma linguagem para outra

linguagem são chamados de tradutores. A linguagem na qual o programa original está expresso é chamada de linguagem fonte e a linguagem para a qual ela será convertida é conhecida como linguagem alvo. Tanto a linguagem fonte quanto a linguagem alvo definem níveis de abstração específicos.

Se existir um processador que possa executar diretamente programas escritos na linguagem fonte, não há necessidade de se traduzir o programa fonte para uma linguagem alvo.

O método de tradução é empregado quando há um processador (seja ele implementado em hardware ou por interpretação) disponível para executar programas expressos na linguagem alvo mas não na linguagem fonte. Se a tradução tiver sido feita corretamente, a execução do programa traduzido vai obter exatamente os mesmos resultados que a execução do programa fonte obteria se houvesse um processador que o executasse diretamente.

É importante observar a diferença entre tradução e interpretação. Na tradução, o programa original, expresso na linguagem fonte, não é executado diretamente. Em vez da execução direta, esse programa precisa ser convertido para um programa equivalente, conhecido como programa objeto ou programa binário executável, que será executado após o término do processo de tradução.

Logo, a tradução envolve dois passos distintos:

- Geração de um programa equivalente na linguagem alvo;
- Execução do programa obtido.

No processo de interpretação existe apenas um único passo: a execução do programa original na linguagem fonte.

Nesta seção iremos analisar a função dos softwares básicos: compilador, montador, ligador, carregador e interpretador. Programas necessários para que os softwares dos usuários implementados em alguma linguagem de programação específica possam ser executados em um computador.

5.3.1 Tradutores

Os tradutores podem ser divididos em dois grupos, dependendo da relação existente entre a linguagem fonte e a linguagem alvo. Quando a linguagem fonte for essencialmente uma representação simbólica para uma linguagem de máquina numérica, o tradutor é chamado de montador e a linguagem fonte é chamada de linguagem de montagem. Quando a linguagem fonte for uma linguagem de alto nível, como é o caso do Pascal ou do C, e a linguagem alvo for uma linguagem de máquina numérica ou uma representação simbólica desta linguagem (linguagem de montagem), o tradutor é chamado de compilador.

Podemos observa na Figura 5.6 [66] todos os passos necessários para que um algoritmo expresso em uma linguagem de programação possa ser carregado em memória para ser executado por um computador. Cada fase possui um conjunto de entradas e saídas de seu processamento. Estas fases e seus respectivos softwares envolvidos são descritas nas seções seguintes.

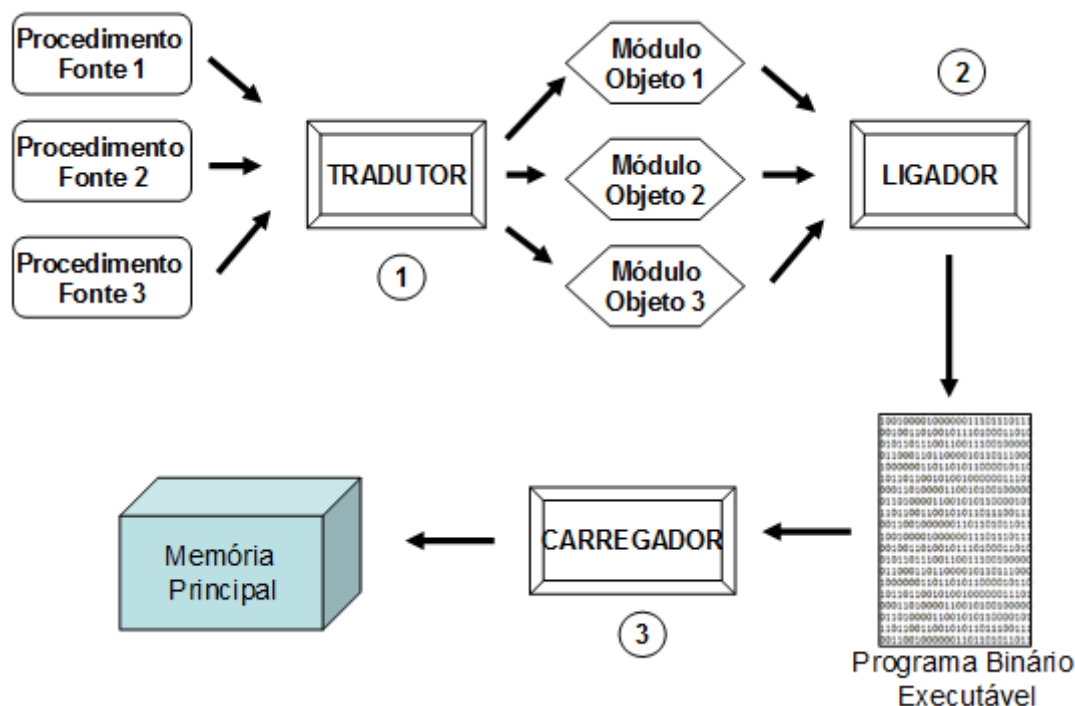



Figura 5.6: Etapas do processo de compilação.

5.3.2 Processo de Compilação

Diferente do processo de montagem de um programa em linguagem de montagem para um programa em linguagem de máquina, que é bastante simples, pois existe um mapeamento direto de um para um entre os comandos em linguagem de montagem e os equivalentes em código binário, o processo de compilação de linguagens é muito mais complexo.

5.3.2.1 Passos da compilação

Considere o comando simples abaixo:

```
A = B + 4;
```

O compilador tem que resolver um número grande de tarefas na conversão deste comando em um ou mais comandos em linguagem de montagem:

1. Reduzir o texto do programa para símbolos básicos da linguagem, como identificadores tais como A e B, demarcações como o valor constante 4 e delimitadores do programa tais como = e +. Esta parte da compilação é chamada de análise léxica.

2. Decodificar os símbolos para reconhecer a estrutura do programa. No comando usado acima, por exemplo, um programa chamado parser deve reconhecer o comando como sendo uma atribuição de valores da forma:

   ```
   <Identificador> "=" <Expressão>
   ```

   onde `<Expressão>` é decodificado na forma:

```
<Identificador> "+" <Constante>
```

Essa tarefa é chamada de análise sintática.

3. Análise de nomes: associar os nomes A e B com variáveis do programa, e associá-los também a posições de memória específicas onde essas variáveis serão armazenadas durante a execução.

4. Análise de tipos: determinar os tipos de todos os dados. No caso anterior, as variáveis A e B e a constante 4 seriam reconhecidas como sendo do tipo int em algumas linguagens. As análises de nome e tipo são também conhecidas como análise semântica: determina o significado dos componentes do programa.

5. Mapeamento de ações e geração de código: associar comandos do programa com uma sequência em linguagem de montagem apropriada. No caso anterior, a sequência em linguagem de montagem poderia ser:

   Comando de atribuição

   ```
   ld[B], %r0, %r1    // Carrege variável B em um registrador.
   add %r1, 4, %r2    // Calcule o valor da expressão.
   st %r2, %r0, [A]   // Faça a atribuição na variável A.
   ```

6. Existem passos adicionais que o compilador deve tomar, tais como, alocar variáveis a registradores, usar registradores e, quando o programador desejar, otimizar o programa. O otimizador de código (independente de máquina) é um módulo opcional (presente na grande maioria dos compiladores) que objetiva melhorar o código intermediário de modo que o programa objeto produzido ao fim da compilação seja menor (ocupe menos espaço de memória) e/ou mais rápido (tenha tempo de execução menor). A saída do otimizador de código é um novo código intermediário.

5.3.3 Processo de Montagem

O processo de traduzir um programa em linguagem de montagem para programa em linguagem de máquina é chamado de processo de montagem. Este processo é muito simples, uma vez que existe um mapeamento um para um de comandos em linguagem de montagem para seus correspondentes em linguagem de máquina. Isto é o contrário da compilação, onde um comando em linguagem de alto nível pode ser traduzido em vários comandos em linguagem de máquina.

5.3.3.1 Por que usar uma Linguagem de Montagem?

Programar em uma linguagem de montagem não é fácil. Além da dificuldade, o desenvolvimento de um programa na linguagem de montagem consome mais tempo do que seu desenvolvimento em uma linguagem de alto nível. A depuração e manutenção dos programas em linguagem de montagem são mais complicados.

Nessas condições, por que alguém escolheria programar em uma linguagem de montagem?

Existem duas razões que justificam esta opção: performance e acesso aos recursos da máquina. Um expert na linguagem de montagem pode produzir um código menor e muito mais eficiente do que o gerado por um programador usando linguagem de alto nível.

Em segundo lugar, certos procedimentos precisam ter acesso total ao hardware. Por exemplo, se a máquina alvo tiver um bit para expressar o overflow de operações aritméticas, um programa em linguagem de montagem pode testar diretamente este bit, coisa que um programa em Java não pode fazer. Além disso, um programa em linguagem de montagem pode executar qualquer uma das instruções do conjunto de instruções da máquina alvo.

5.3.3.2 Tarefas do montador

Embora a montagem seja um processo simples, é tedioso e passível de erros quando feito manualmente. Montadores comerciais têm ao menos as seguintes características:

- Permitem ao programador especificar posição de valores de dados e programas durante a execução;
- Permitem que o programador de início realize valores de dados na memória antes da execução do programa;
- Implementam mnemônicos em linguagem de montagem para todas as instruções da máquina e modos de endereçamento, e traduzem comandos em linguagem de montagem válidos, nos seus equivalentes em linguagem de máquina;
- Permitem o uso de rótulos simbólicos para representar endereços e constantes;
- Incluem um mecanismo que permite que variáveis sejam definidas em um programa em linguagem de montagem e usadas em outros programas separadamente;
- Possibilitam a expansão de macros, ou seja, rotinas (semelhantes às funções em linguagem de alto nível) que podem ser definidas uma vez e então instanciadas quantas vezes necessário.

5.3.3.3 Montadores de dois passos

A maioria dos montadores leem textos do programa em linguagem de montagem duas vezes, e são chamados de “montadores de dois passos”. O primeiro passo serve para determinar o endereço de todos os itens de dados e instruções de máquina, e selecionar quais instruções devem ser geradas para cada instrução em linguagem de montagem (mais ainda não gerá-las).

Os endereços dos itens de dados e instruções são determinados por meio do uso de um contador de programa para a montagem, chamado contador de localização. O contador de localização gerencia o endereço da instrução executada e dos itens de dados durante a montagem, que geralmente é inicializada com `0` (zero). No início do primeiro passo, é incrementado de acordo com o tamanho de cada instrução.

Durante este passo, o montador também efetua quaisquer operações aritméticas em tempo de montagem, e insere as definições de todos os rótulos de funções e variáveis e as constantes, em uma tabela chamada Tabela de Símbolos.

A razão principal para exigir uma segunda passagem é permitir que símbolos sejam usados no programa antes de serem definidos. Após a primeira passagem, o montador terá identificado todos os símbolos e os colocado na Tabela de Símbolos, já durante a segunda passagem, gerará código de máquina, inserindo os identificadores dos símbolos que agora são conhecidos.

5.3.4 Ligação e Carregamento

A maioria dos programas é composto de mais de um procedimento. Os compiladores e os montadores geralmente traduzem um procedimento de cada vez, colocando a saída da tradução em disco. Antes que o programa possa rodar, todos os seus procedimentos precisam ser localizados e ligados uns aos outros de maneira a formarem um único código.

5.3.4.1 Ligação

A função do ligador é coletar procedimentos traduzidos separadamente e ligá-los uns aos outros para que eles possam executar como uma unidade chamada programa binário executável.

Se o compilador ou o montador lesse um conjunto de procedimentos fonte e produzisse diretamente um programa em linguagem de máquina pronto para ser executado, bastaria que um único comando fonte fosse alterado para que todos os procedimentos fonte tivessem que ser novamente traduzidos.

Usando a técnica do módulo objeto separado, o único procedimento a ser novamente traduzido seria aquele modificado. Havendo a necessidade de realizar apenas a etapa de ligação dos módulos separados novamente, sendo esta tarefa mais rápida que a tradução.

5.3.4.2 Carregamento

O carregador é um programa que coloca um módulo de carregamento na memória principal. Conceitualmente, a tarefa do carregador não é difícil. Ele deve carregar os vários segmentos de memória com seus valores corretos e inicializar certos registradores, tais como o apontador para pilha do sistema, responsável pelo escopo das rotinas que estarão em execução e o contador de instruções contido no processador, com seus valores iniciais, indicando assim onde o programa deve ser iniciado.

Em Sistemas Operacionais modernos, vários programas estão residentes na memória a todo instante, e não há como o montador ou o ligador saber em quais endereços os módulos de um programa irão residir. O carregador deve relocar estes módulos durante o carregamento adicionando um deslocamento a todos os endereços, permitindo desta forma acessar cada módulo individualmente na memória.

Esse tipo de carregamento é chamado de carregamento com relocação. O carregador simplesmente modifica endereços relocáveis dentro de um único módulo de carregamento para que vários programas passem a residir na memória simultaneamente.

5.4 Interpretadores

O software interpretador é um programa de computador que executa instruções escritas em uma linguagem de programação. Por exemplo, as linguagens Basic, Prolog, Python e Java, são frequentemente interpretados. Um interpretador geralmente usa uma das seguintes estratégias para a execução do programa: executar o código fonte diretamente ou traduzir o código fonte em alguma eficiente representação intermediária e depois executar este código.

Para isso, certos tipos de tradutores transformam uma linguagem fonte em uma linguagem simplificada, chamada de código intermediário, que pode ser diretamente "executado" por um programa chamado interpretador. Nós podemos imaginar o código intermediário como uma linguagem de máquina de um computador abstrato projetado para executar o código fonte.

Interpretadores são, em geral, menores que compiladores e facilitam a implementação de construções complexas em linguagens de programação. Entretanto, o tempo de execução de um programa interpretado é geralmente maior que o tempo de execução deste mesmo programa compilado, pois o interpretador deve analisar cada declaração no programa a cada vez que é executado e depois executar a ação desejada, enquanto que o código compilado apenas executa a ação dentro de um contexto fixo, anteriormente determinado pela compilação. Este tempo no processo de análise é conhecido como "overhead interpretativa".

5.5 Usando os Softwares Básicos

A ferramenta `cc` é um ambiente completo para a compilação, montagem e ligação dos programas de usuário desenvolvidos na linguagem C. Nesta prática iremos identificar cada passo no processo de transformar um código de alto nível (Linguagem C) em um arquivo binário executável.

Nota

O `cc` é um compilador da GNU utilizado principalmente no sistem operacional Linux ou de tipo Unix. Para executá-lo você precisa abrir um terminal e escrever os comandos indicados nesta prática. Você já deve conhecê-lo da disciplina *Introdução a Programação*.

Nota

Lembre-se de os códigos fontes do livro estão disponíveis para download e as intruções de como baixá-los estão em “Baixando os códigos fontes” [x].

Passo 1

Escreva em qualquer editor de texto os seguintes textos e salve com o nome sugerido para cada arquivo:

code/tradutor/teste.h

```
//Soma números
int soma(int op1, int op2);

//Subtrai numeros
int subtrai(int op1, int op2);
```

code/tradutor/teste.c

```
#include "teste.h"

int soma(int op1, int op2)
{
  return op1 + op2;
}

int subtrai(int op1, int op2)
{
  return op1 - op2;
}
```

code/tradutor/main.c

```
#include "teste.h"
#include <stdio.h>

int main(){
  int a, b;
  a = soma(2, 3);
  printf("Soma = %d\n", a);
  b = subtrai(4, 3);
  printf("Subtração = %d\n", b);
}
```

Dica

Caso você tenha baixado o código deste livro, basta entrar na pasta a seguir onde estes arquivos já existem:

`livro/capitulos/code/tradutor`

Passo 2

Usar o Compilador do `cc`. Ele irá processar cada arquivo .c e gerar um arquivo .s com o código Assembly respectivo.

Linguagem Fonte: Linguagem C
Linguagem Alvo: Assembly

Comando

```
cc -S main.c teste.c
```

Passo 3

Usar o montador da aplicação `cc`. Ele irá processar cada arquivo .s e gerar um arquivo .o com o código objeto respectivo.

Linguagem Fonte: Assembly
Linguagem Alvo: Código Objeto

Comando

```
cc -c main.s teste.s
```

Os passos 2 e 3 geralmente são realizados juntos, para isso basta realizar o seguinte comando: `cc -c main.c teste.c`. Gerando assim um arquivo com Código Objeto (.o) para cada arquivo .c, pulando o código assembly (código intermediário).

Passo 4

Usar o ligador do `cc`. Ele irá processar todos os arquivos .o e ligá-los em um único arquivo binário executável.

Linguagem Fonte: Código Objeto
Linguagem Alvo: Código Binário (Executável)

Comando

```
cc main.o teste.o -o exec
```

Nota

`exec` foi o nome dado ao arquivo executável gerado pelo ligador do `cc`, logo, o usuário pode escolher qualquer nome para o mesmo.

Passo 5

Executar o Código Executável. No linux, para carregar um código executável e colocá-lo em execução basta seguir o seguinte comando:

Comando

```
./exec
```

Passo 6

Modificar o código do arquivo teste.c .

EDITAR A IMPLEMENTAÇÃO DA FUNÇÃO SUBTRAI.

Trocar: `return op1 - op2;`

Por: `return op1 + op2;`

Nota

O conteúdo do arquivo `teste.c` ficará igual a:

code/tradutor/teste.c

```c
#include "teste.h"

int soma(int op1, int op2)
{
  return op1 + op2;
}

int subtrai(int op1, int op2)
{
  return op1 + op2;
}
```

Passo 7

Compilar apenas o arquivo teste.c

Comando

```
cc -c teste.c
```

Passo 8

Ligar os códigos objetos novamente.

Comando

```
cc main.o teste.o -o exec
```

Passo 9
Executar o novo Código Executável. Observar a diferença das execuções.

Comando
./exec

Importante
Podemos observar a utilidade do software ligador, pois após o **Passo 6** não há necessidade de recompilar todos os códigos fonte, apenas o arquivo `teste.c`, gerando assim um novo arquivo objeto `teste.o` que será ligado ao arquivo `main.o` (arquivo não modificado) formando o novo código executável `exec`.

Dica
Você pode assistir esta prática no seguinte vídeo:

Figura 5.7: Prática de softwares básicos: http://youtu.be/dVL3XQFNY8o

5.6 Recapitulando

Neste capítulo estudamos o conceito de algoritmo e vimos que o mesmo pode ser implementado por diversos mecanismos mecânicos e elétricos. Um algoritmo é descrito em um computador através de uma linguagem de programação. São diversos os níveis de abstração em cada linguagem, cada uma com um objetivo distinto. Para que todas estas linguagens possam coexistir no computador foram criados software básicos com o objetivo de realizar a execução do algoritmo descrito através destas linguagens de programação.

Dentre os softwares básicos estudados vimos os Tradutores e Interpretadores, cada um com seu uso exclusivo. Os Tradutores ainda podem ser classificados em Compiladores e Montadores, ambos tendo como objetivo traduzir uma linguagem fonte para uma linguagem alvo cujo Interpretador seja implementado no computador corrente.

5.7 Atividades

1. Quais os ganhos que as linguagens de programação de alto nível trazem para os programadores?

2. Descreva 3 diferentes paradigmas de programação.
3. Sistemas Operacionais são tipos de software básico? Quais os tipos de softwares básicos existentes?
4. Marque a alternativa correta. Um interpretador, a partir de um programa-fonte:
    a. Gera um programa-objeto para posterior execução
    b. Efetua a tradução para uma linguagem de mais alto nível
    c. Interpreta erros de lógica
    d. Executa instrução a instrução, sem gerar um programa-objeto
    e. Não detecta erros de sintaxe
5. Qual a função de uma linguagem de montagem (linguagem assembly)?
6. Quais as diferenças entre software interpretador e software tradutor?
7. O compilador e o montador são softwares tradutores. Qual a diferença entre eles?

Capítulo 6

Índice Remissivo